NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN IMPLEMENTATION OF REMOTE APPLICATION SUPPORT IN A MULTILEVEL ENVIRONMENT

by

Melissa K. Egan

March 2006

Thesis Advisor:	Cynthia E. Irvine
Thesis Co-Advisor:	Thuy D. Nguyen

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Implementation of Remote Application Support in a Multilevel Environment			**5. FUNDING NUMBERS**
6. AUTHOR(S) Melissa K. Egan			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

There is a growing need for high-assurance architectures that support mandatory confidentiality and integrity policies. One such architecture currently under development is the Monterey Security Architecture (MYSEA), a distributed multilevel secure (MLS) computing environment that integrates untrusted commercial off-the-shelf components with specialized high-assurance elements.

To ensure that information is purged from untrusted client PCs between sessions at different security levels, MYSEA clients are diskless. Therefore, it is desirable for thin MYSEA clients to be able to remotely execute server-resident applications, which may in turn request access to data residing elsewhere on the MLS Local Area Network (LAN). This functionality must be implemented in such a way that the access control policies of the multilevel environment are maintained. Working from a detailed design for remote application support, this thesis involved the implementation and testing of the remote application support functionality. Beyond the implementation of remote application support itself, this thesis involved the porting of a Trivial File Transfer Protocol (TFTP) client and the development of a simple web client as proof-of-concept remote applications, as well as the creation of a Common Gateway Interface (CGI) mechanism for invoking those remote applications from a client web browser. This research is relevant to the DoD Global Information Grid's vision of assured information sharing.

14. SUBJECT TERMS Multilevel Security (MLS), Information Assurance (IA), Monterey Security Architecture (MYSEA), Remote Application, Trusted Remote Session Management			**15. NUMBER OF PAGES** 148
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

AN IMPLEMENTATION OF REMOTE APPLICATION SUPPORT IN A MULTILEVEL ENVIRONMENT

Melissa K. Egan
Civilian, Naval Postgraduate School
B.A., Pomona College, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2006

Author: Melissa K. Egan

Approved by: Cynthia E. Irvine, Ph.D.
Thesis Advisor

Thuy D. Nguyen
Co-Advisor

Peter J. Denning, Ph.D.
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

There is a growing need for high-assurance architectures that support mandatory confidentiality and integrity policies. One such architecture currently under development is the Monterey Security Architecture (MYSEA), a distributed multilevel secure (MLS) computing environment that integrates untrusted commercial off-the-shelf components with specialized high-assurance elements.

To ensure that information is purged from untrusted client PCs between sessions at different security levels, MYSEA clients are diskless. Therefore, it is desirable for thin MYSEA clients to be able to remotely execute server-resident applications, which may in turn request access to data residing elsewhere on the MLS Local Area Network (LAN). This functionality must be implemented in such a way that the access control policies of the multilevel environment are maintained. Working from a detailed design for remote application support, this thesis involved the implementation and testing of the remote application support functionality. Beyond the implementation of remote application support itself, this thesis involved the porting of a Trivial File Transfer Protocol (TFTP) client and the development of a simple web client as proof-of-concept remote applications, as well as the creation of a Common Gateway Interface (CGI) mechanism for invoking those remote applications from a client web browser. This research is relevant to the DoD Global Information Grid's vision of assured information sharing.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ABBREVIATIONS AND ACRONYMS

CGI	Common Gateway Interface
DAC	Discretionary Access Control
COTS	Commercial Off-The-Shelf
I&A	Identification and Authentication
LAN	Local Area Network
MAC	Mandatory Access Control
MLS	Multilevel Secure
MYSEA	Monterey Security Architecture
OS	Operating System
RA	Remote Application
SAK	Secure Attention Key
SSD	Secure Session Daemon
SSS	Secure Session Server
STOP	Secure Trusted Operating Program
TCM	Trusted Channel Module
TFTP	Trivial File Transfer Protocol
TPE	Trusted Path Extension
TPS	Trusted Path Server
TRSS	Trusted Remote Session Server
URL	Uniform Resource Locator

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. Cynthia Irvine and Prof. Thuy Nguyen, for the time, effort, and guidance they have provided throughout this project. I would also like to thank Jean Khosalim and David Shifflett for their technical expertise and assistance.

This material is based upon work supported by the National Science Foundation under grant No. DUE-0114018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION OF STUDY

Government, military, and business organizations that currently maintain multiple networks to host data at differing security levels stand to benefit by replacing them with a single high-assurance multilevel secure (MLS) network, due to the increased efficiency and reduced costs associated with the management of a single network as opposed to many. There is a growing need for high-assurance architectures that implement multi-domain information protection mechanisms.

The Global Information Grid (GIG) envisioned by the United States Department of Defense provides "a seamless, secure, and interconnected information environment, meeting real-time and near real-time needs of both the warfighter and the business user" [1]. This environment will likely contain data residing at various security levels, and could therefore benefit from the use of cross-domain solutions that would enable multiple levels of data to be accessed from a single node. One such solution currently under development is the Monterey Security Architecture (MYSEA), a distributed multilevel secure computing environment that integrates untrusted commercial off-the-shelf components with specialized high-assurance elements [2].

In the MYSEA architecture, multilevel servers provide isolation between security domains and thin clients access the data within each domain as permitted by the servers. To ensure that information is purged from untrusted client PCs between sessions at different security levels, MYSEA clients are diskless. Therefore, it is desirable for thin MYSEA clients to be able to remotely execute server-resident applications, which may in turn request access to data residing elsewhere on the MLS Local Area Network (LAN) or on a connected single-level network. This functionality must be implemented in such a way that the access control policies of the multilevel environment are maintained. Prior to the completion of this thesis, the design for remote application support had been written [3], but not yet implemented.

B. PURPOSE OF STUDY

The objective of this thesis was to implement remote application support in the MYSEA environment. Implementation was to be based on the pre-existing design, refined as necessary during the course of implementation and testing. This thesis was meant to provide concrete answers to the questions:

1. What modifications to the existing design are necessary to successfully implement remote application support on the MYSEA server?
2. What additional functionality, if any, is required to support the remote execution of specific desired applications?

Beyond the implementation of the trusted server processes that enable remote application support, this project involved the porting of two simple network-enabled remote applications onto the MYSEA server, and the creation of a Common Gateway Interface (CGI) mechanism for invoking those remote applications from a client web browser. This project further involved the development and execution of a test plan to verify the functionality of each of the newly implemented remote application support components and of the integrated remote application support system.

C. ORGANIZATION OF THESIS

This thesis is organized as follows:

- Chapter I has provided an introduction to the motivation and purpose of this thesis, including a brief introduction to the Monterey Security Architecture (MYSEA) and the anticipated role of remote application support in the multilevel MYSEA environment.
- Chapter II provides a more in-depth look at the MYSEA architecture and the multilevel XTS-400 system that hosts the MYSEA server software, focusing on those aspects of its architecture that have most influenced the design for remote application support. This chapter also describes the envisioned functionality and benefits brought about by the implementation of remote application support in the MYSEA environment.

- Chapter III provides a high-level description of the requirements and design for remote application support, as described in a previous thesis [3]. This chapter also discusses changes made to the design between the publication of [3] and the beginning of this project, and specifies which of the remote application support components laid out in [3] were modified or implemented for this project.
- Chapter IV describes the development environment, implementation requirements for each module developed as part of this project, and unforeseen issues that arose during the course of implementation.
- Chapter V describes the developmental and acceptance testing performed on each of the components implemented for this project.
- Chapter VI concludes with a project summary and suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. MONTEREY SECURITY ARCHITECTURE (MYSEA)

The Monterey Security Architecture (MYSEA), under development at the Naval Postgraduate School, was designed to serve as a trusted, distributed, multilevel secure (MLS) environment that integrates a small number of high-assurance components with a greater number of untrusted commercial off-the-shelf (COTS) components. The motivation behind this architecture was the realization that "unless a secure system offers users comfortable and familiar interfaces for handling routine information, it will fail due to lack of user acceptability" [2]. The MYSEA architecture consists of three types of physical components: high-assurance MLS servers; commercial off-the-shelf client PCs; and pre-existing single-level networks.

The MYSEA server runs on a DigitalNet XTS-400, a high-assurance multilevel secure platform that includes a Trusted Computing Base (TCB) operating in conjunction with untrusted, constrained Application Protocol Servers. The MYSEA server consists of the following additional components: a Secure Session Services (SSS) component that is used to launch instances of the untrusted application protocol servers at the current session levels of the users requesting their services; a Trusted Path Services (TPS) component, which extends the native XTS-400 trusted path support for local terminals to remote MYSEA clients; and a Dynamic Security Services (DSS) Manager[1], which governs security and performance factors of the various MYSEA components.

A MYSEA client consists of two physical components: an untrusted personal computer (PC) and a Trusted Path Extension (TPE). The PC is a COTS product, either a thin client that accesses remote applications or a typical PC hosting a commercial operating system and application suite. The TPE, which currently has a hand-held form factor, is responsible for providing a secure, unforgeable connection between the user and the security functions of the MYSEA server by way of a trusted path. The trusted path is invoked when the user presses the Secure Attention Key (SAK) on the client.

[1] This component is currently under development.

To ensure that session information is purged from untrusted client PCs between sessions at different security levels, MYSEA clients are stateless. On startup, they boot their operating system (for example, a Linux or Windows variant) from a non-writable source into RAM. The use of popular commercial operating systems on the client systems ensures that users may continue to use their favorite applications with an interface they are familiar with. It also makes the acquisition of end-user applications much easier in terms of cost and availability.

MYSEA servers and clients are co-located on a MLS local area network (LAN), and the network interface on the server connecting it to the MYSEA clients is configured at the level of the MLS LAN. This means that any process running on the server that requires access to the network interface, including the SSS and TPS processes, must also run at the level of the MLS LAN. This requirement will be revisited later in this chapter.

One or more single-level networks, each operating at its own security level, may be connected to the MYSEA server by way of a Trusted Channel Module (TCM). The function of the TCM is to authenticate the single-level network to the MYSEA server and provide high-assurance labeling of the information entering the server from the network.

B. XTS-400 ARCHITECTURE

1. Overview

DigitalNet's XTS-400 system, the platform on which the MYSEA server is based, combines an Intel x86 hardware base with the high-assurance multilevel secure STOP 6 operating system. STOP provides a Linux-like user and programming interface, allowing for many existing Linux applications to be run on the XTS without modification. It provides not only traditional discretionary access control (DAC), but also mandatory access control (MAC), which is implemented in accordance with the Bell-LaPadula model for confidentiality [4] and the Biba policy for integrity [5].

The Bell-LaPadula security model [4] formalizes a policy to prevent unauthorized access to confidential data. It does this by enforcing two policies: the simple security policy and the security-* (or confinement) property. The simple security policy states that a subject may read an object if and only if the security level of the subject dominates

(is greater than or equal to) the security level of the object. This would allow, for example, a user logged in at the Secret level to read Secret, Confidential, and Unclassified documents, but not Top Secret documents. The * property states that a subject may write to an object if and only if the security level of the subject is dominated by the security level of the object. This would allow a user logged in at the Secret level to write to Secret and Top Secret documents, but not to Confidential or Unclassified documents. The STOP implementation of the Bell-LaPadula model actually enforces a stricter version of this rule: it allows a subject to write to an object only when the subject and object are at the same security level. This prevents lower-level subjects from writing higher-level objects they cannot later access [6].

The Biba integrity model [5] was designed to prevent the unauthorized modification of data. It does this by enforcing two analogous policies for integrity: the simple integrity policy and the integrity-* property. The simple integrity policy states that a user may read an object if and only if the integrity level of the object dominates the integrity level of the subject. The integrity-* property states that a subject may write to an object if and only if the integrity level of the subject dominates the integrity level of the object. (Once again, STOP implements a variant of this policy that allows writes by subjects only to objects at the same integrity level.) Note that these policies are equivalent to the Bell-LaPadula security policies with the dominance relations reversed. The STOP system's mandatory integrity policy allows for the establishment of highly protected execution domains in which executables may read files they need, while those files remain protected from modification by unauthorized logic or malicious code [6].

The STOP operating system distinguishes between applications that are trusted and those that are untrusted. System administrators may grant applications certain privileges, including the ability to bypass mandatory and/or discretionary controls. Any application granted such privileges or assigned a high integrity level is considered a *trusted application*. All other applications, including (by default) the user commands and tools familiar to Linux/UNIX users, are considered untrusted and are subject to the MAC and DAC policies described above [6].

The XTS-400 has been evaluated at the Common Criteria [7] EAL-5+ level [8]. Its access control mechanisms, high level of assurance, and support for Linux applications make it a suitable platform for the MYSEA server, and several of its unique features have helped shape the design for remote application support. These features, and their influence on the design, are described in [3] and summarized in the following sections.

2. TCP/IP Privileged Ports

The XTS prohibits normal user programs from using TCP/IP privileged ports; only processes running as the *network* user can use these ports. Furthermore, only trusted programs may be given the privilege to change their user identifier.

Remote applications and application protocol servers need to make use of privileged ports, but are designed to run at the level of the user invoking them. To designate them as trusted applications would be undesirable, as it would increase the amount of trusted code necessary on the system. The restriction on privileged port access therefore necessitates the introduction of trusted programs to service these ports on behalf of untrusted remote applications and application protocol servers.

3. Trusted and Untrusted Processes

The processes implementing remote application support require MAC/DAC exemption in order to communicate with both the network interface (which runs at the level of the MLS LAN) and the untrusted remote application (which runs at the level of the user who invoked it). Certain processes also require the ability to set their user ID to that of the *network* user, as discussed in the previous section. In order to be granted these privileges, the programs must be designated trusted applications. Privilege requirements specific to each process are discussed in more detail in Chapter III.

4. System Calls

Many of the standard Linux system calls are available on the XTS-400 and are accessible to user applications. However, some system calls are specific to the XTS-400

and may only be used by trusted applications. Others have increased functionality when called by trusted applications. For example, a *read* call may provide the capability to read an object at any of various levels when invoked by a trusted process [3].

Processes implementing remote application support make use of XTS-specific system calls in order to enable and disable any special privilege sets that they require. A Privileges Module, discussed further in Chapter III, has been implemented to encapsulate these system calls in an intuitive interface. Per-process privilege requirements are also discussed in Chapter III.

C. OVERVIEW OF REMOTE APPLICATION SUPPORT

With the implementation of remote application support, a user logged in to the MYSEA server remotely has the ability to launch server-resident applications from his or her client using only a web browser. This is a useful capability, considering that MYSEA clients are diskless and must therefore work around the following limitations:

1. The amount of RAM available to run applications on the client may be limited.
2. Data created or retrieved by the client will be purged at the end of each user session.
3. Typical system maintenance tasks involving the installation or upgrade of client software will be complicated by the client's lack of non-volatile storage. Installing or upgrading a single application could require re-burning the entire non-writeable medium on which the client's operating system and complete application suite are stored.

In a remote-application-enabled environment, a user logged in to a MYSEA server from a remote client may use a simple web interface to request a list of remote applications supported by the server or to specify a particular remote application to be executed. The requested remote application is invoked on the server, and its output is displayed back to the user via the web interface. Because the processing is performed on the server, the user need not worry whether sufficient resources are available on the

client, and data retrieved or created by the user may be saved in persistent storage on the server. Furthermore, the centralized server-resident applications may be easily configured and updated, and are accessible to users logged in from MYSEA clients of any platform.

D. SUMMARY

This chapter has provided an introduction to the MYSEA architecture, including the security policies enforced within the multilevel MYSEA environment and specific characteristics of the XTS-400 system on which the MYSEA server software is installed. It has introduced the concept of remote application support, and discussed the envisioned benefits that remote application support will bring to the MYSEA environment.

Chapter III more formally describes the functional requirements for remote application support, and summarizes the detailed design specifications [3] that paved the way for the implementation of remote application support on the MYSEA server.

III. REQUIREMENTS AND DESIGN

A. INTRODUCTION

The remote application functionality described in the previous chapter must be implemented in such a way that the access control policies of the multilevel environment are maintained. This means, for one, ensuring that remote applications are launched at the session level of the user invoking them. It also means ensuring that applications launched remotely are able to establish outbound connections only with those destinations authorized to communicate at the application's current session level. These policies must be enforced while operating within the constraints imposed by the STOP operating system regarding trusted and untrusted processes, privileged ports, and privileged systems calls, while minimizing the amount of trusted code that must be introduced onto the system.

This chapter describes, at a high level, the design for MYSEA remote application support that was developed and presented in a previous thesis [3]. Section B of this chapter lists the functional requirements that shaped this design. Section C describes each of the processes, databases, and modules contributing to the functionality of the remote application support mechanism, including those that were implemented both prior to and during the course of this project. Section D describes changes made to the design subsequent to its original specification in [3] but prior to its implementation. Finally, Section E identifies those components of the remote application functionality that needed to be implemented or modified specifically for this project.

B. TOP-LEVEL REQUIREMENTS

The design for remote application (RA) support was derived from the following top-level user requirements [3]:

- The RA shall be able to communicate with the local MLS server, a remote MLS server and a RA server.

- The remote application shall be appropriately bound to the authenticated user's session. Specifically, the remote application process shall run with the user ID and at the current session level of the authenticated user.
- The user shall be able to launch the RA from the client.
- The MLS server shall be able to support both Unix/Linux and Microsoft Windows clients.
- The design shall only require a minimal number of changes to the RA.

C. HIGH-LEVEL DESIGN

1. Overview

Figure 1 presents an overview of the processes playing a part in remote application support on the MYSEA server. Shaded boxes represent processes that were implemented specifically for this project.

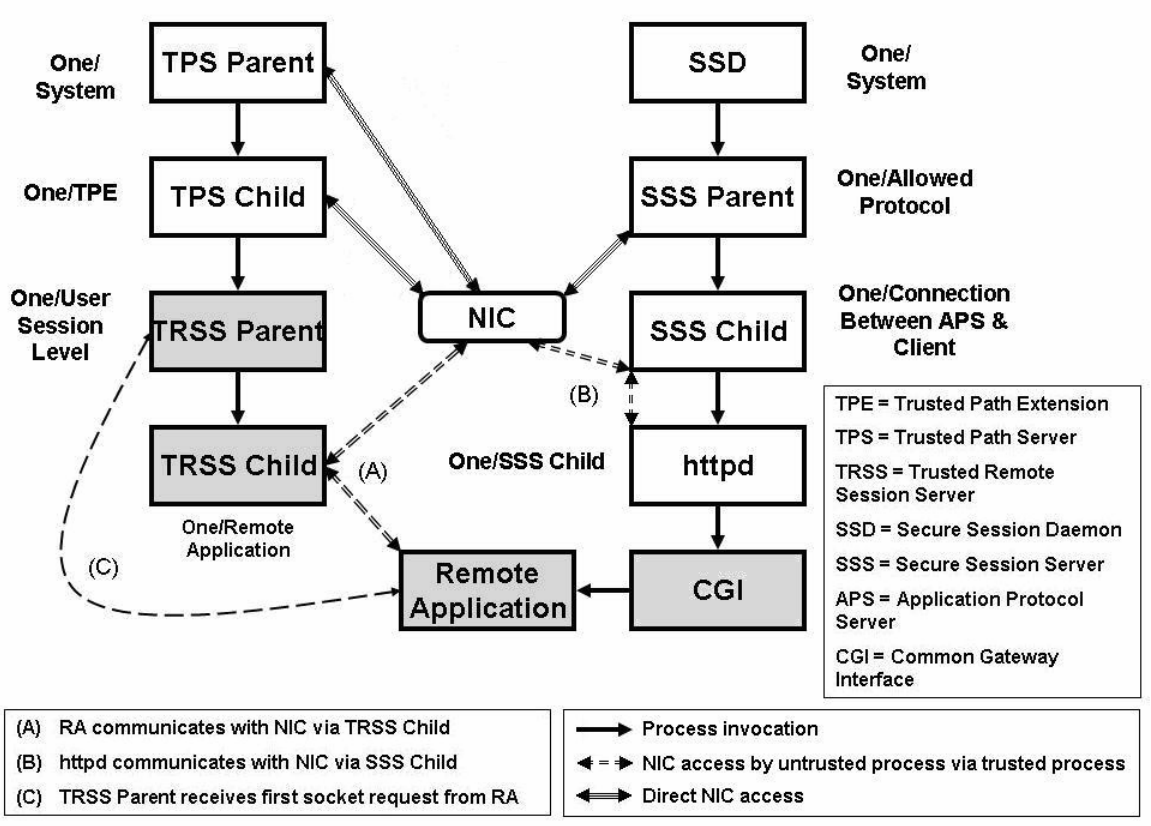

Figure 1. Process Overview (after [3])

The Secure System Daemon (SSD) and Trusted Path Server (TPS) Parent processes are system daemons launched at system startup. The SSD launches a Secure Session Server (SSS) Parent for each application protocol supported by the server, including HTTP. The TPS Parent accepts login requests from remote clients, and delegates the handling of trusted path communications from each client to a dedicated TPS Child process. If the user successfully logs in, the TPS Child creates a Trusted Remote Session Server (TRSS) Parent to handle remote application requests from that user. The SSD, SSS, TPS, and TRSS processes are each considered trusted, and are allowed access to the MLS network interface.

After successfully logging in from a Trusted Path Extension (TPE), a user may open up a web browser and request a web page with links to launch supported remote applications. The SSS Parent designated to handle HTTP requests receives the user's request and passes it to an SSS Child designated to handle HTTP requests specifically from that user. The SSS Child in turn launches httpd, the HTTP daemon, which handles the user's web page request. Because httpd is an untrusted application and cannot access the MLS network interface directly, nor bind to privileged ports, all of its socket communications must go through the trusted SSS Child process that launched it.

The user selects a remote application from the list, and *httpd* executes a Common Gateway Interface (CGI) script which launches the application at the session level of the requesting user. This application is also untrusted and cannot access the MLS network interface directly. If the application requires the use of sockets, it signals the TRSS Parent, which launches a trusted, dedicated TRSS Child process to handle socket communications on behalf of the untrusted application. These may include communications with peers residing elsewhere on the network, or with an Application Protocol Server (APS) residing on the local server. In the latter case, the resulting APS connection request must be validated by the SSS Parent, which launches an SSS Child to handle requests specifically from the requesting remote application to the target APS.

The aforementioned processes are described in more detail in the following sections.

2. Processes

a. Trusted Path Server (TPS) Parent

The Trusted Path Server Parent process is a system daemon that runs at the startup of the MYSEA server. It is responsible for accepting and validating trusted path session requests received from TPEs. When it receives a request for a trusted path session, it checks for the TPE identifier in the Allowed TPE Database. If the TPE identifier is found, the TPS Parent forks a TPS Child process to handle the trusted path connection with the TPE. Otherwise, it drops the connection. It then continues to accept and validate subsequent TPS requests.

The TPS Parent runs at the level of the MLS LAN. Because it must also write to system-low databases, it is designated a trusted process.

b. Trusted Path Server (TPS) Child

Once a TPE request has been validated by the TPS Parent, the TPS Child is responsible for handling further trusted path session communications from that TPE. This includes requests to log in, set the session level, run the session, and log out. When the TPS Child receives data from the client, it first checks that it begins with a Secure Attention Request Packet (SARP). It then reads and processes the client's command, transmits the output to the client, and waits for the next command.

The TPS Child process is also responsible for launching a TRSS Parent process to handle connection requests by remote applications launched from the client.

The TPS Child runs at the level of the MLS LAN. Because it also requires access to system-low databases, as well as the system's user identification and authentication information, it is designated a trusted process.

c. Secure Session Daemon (SSD)

The Secure Session Daemon is a system daemon that runs at startup. It first checks the Allowed Protocols Database to see which protocols the server supports (e.g., HTTP, IMAP, SMTP, etc.). For each protocol entry in the database, it then forks an SSS Parent process to handle service requests for that protocol.

Before forking, it obtains the handles to the User Database and Remote Connection Database so that each of its forked child processes will not have to.

The SSD process terminates when either, all of its forked children have terminated, or it is interrupted by a local Secure Attention Key (SAK). In the latter case, it is responsible for also signaling each of its forked SSS Parent processes to terminate. Because these processes require access to privileged ports and must therefore run as the *network* user, the SSD process must also be able to run as the *network* user in order to fork and terminate them. The SSD is therefore designated a trusted process.

d. Secure Session Server (SSS) Parent

Each Secure Session Server Parent process is responsible for accepting and validating application protocol service requests for a particular TCP/IP protocol. After accepting a request, the process verifies that a trusted session has been established for the requesting TPE by checking to see whether it has an entry in the User Database. If no such entry is found, it checks for a valid remote application connection in the Remote Connections Database. If the TPE is found to be associated with either a valid user session or a valid remote application connection, an SSS Child process is spawned to service the connection; otherwise, the connection is dropped. The SSS Parent process then continues to accept and validate new application protocol service requests.

The SSS Parent runs at the level of the MLS LAN. Because it must be able to switch to the user ID of the *network* user in order to access privileged ports, and because it must also be able to fork child processes with MAC/DAC exemption and the ability to change user IDs, it is designated a trusted process.

e. Secure Session Server (SSS) Child

The SSS Child process is responsible for launching an Application Protocol Server (APS) process to service each application protocol service request from a given TPE or remote application. The APS process runs at the session level negotiated by the user via the TPE, or at the session level of the remote application requesting service. Because the APS process is untrusted, it does not have access to the MLS interface and must therefore rely on the SSS Child process to perform socket operations on its behalf. The two processes communicate via a MYSEA socket (MSKT) allocated by the TPS Child process. Any time the APS needs to make a socket call, it enters its request into the MSKT Database and signals the SSS Child process. The SSS Child process then retrieves the requested call type and corresponding parameters from the MSKT Database and makes the appropriate socket library function call on behalf of the APS. Once the function call has completed, the SSS Child sets the return value of the call and any data received into the appropriate fields in the database entry, and signals the APS process. The APS is then able to retrieve the results of its socket call from the database.

When an APS process requests access to a peer (via the *connect, sendto, accept,* or *recvfrom* socket calls), the SSS Child process is responsible for checking whether the connection is allowed before proceeding with the call. It does this by consulting the Peer Level and User Databases to retrieve the current session level of the peer. By comparing the session level of the peer with the session level of the APS, it is able to determine whether the access is allowable. If so, the SSS Child must consult the Source Address Binding Database to determine the source address configured for use with the supplied destination address before binding the socket to a source port/address pair.

For each new connection established on behalf of the APS process, the SSS Child updates the Remote Connections Database with the new connection and its current session level. Upon the closing of each connection, the SSS Child removes the connection from the database.

The SSS Child runs at the level of the MLS LAN; however, it must also communicate with the APS running at the session level of the remote user. Furthermore, it must switch to the user ID of the remote user before initializing the MSKT Database and launching the APS process, so that the APS process runs with the correct user ID and has read/write access to the MSKT database. For these reasons, the SSS Child is designated a trusted process.

f. Application Protocol Server (APS)

The Application Protocol Server handles the server side of a client/server protocol. APS processes are untrusted, and run at the level of the user or remote application that invoked them. APS processes requiring access to the MLS LAN or to privileged ports must request socket operations by way of an MSKT managed by an SSS Child process. If an APS is ported from another platform, its socket calls for these special operations must therefore be modified to use the MSKT interface. A number of APSs have already been ported to the MYSEA server in previous works. These include an Internet Message Access Protocol (IMAP) mail server [9, 10], a Simple Mail Transfer Protocol (SMTP) mail server, viz., Sendmail [11], and a Hypertext Transfer Protocol (HTTP) web server [12].

g. CGI Remote Application Invocation Process

The CGI RA invocation process is spawned by the web APS in response to a user's request to invoke an RA via the web interface. Its primary responsibilities are to process the user's RA request, launch the appropriate RA, and package the output of the RA into a web page to be returned to the user by the web APS.

The RA invocation process is untrusted, and runs with the user ID and at the session level of the invoking APS (and therefore, of the user requesting the RA invocation).

h. Trusted Remote Session Server (TRSS) Parent

The TRSS Parent process is launched by the TPS Child process, and is responsible for spawning a TRSS Child process for each remote application that requests the use of MYSEA sockets.

Upon being launched, the TRSS Parent process first attaches to the databases required for its own operations and for those of its children. It then waits until it is signaled by a remote application requesting a socket call for the first time. Once signaled, it searches the MSKT Database for an entry whose call type is set to *new socket call*. It then forks a TRSS Child process to handle the new socket request and all future socket calls from this remote application, and sets the *trusted PID* field of the MSKT Database entry to the process ID of child it has just forked. It then waits until signaled by the next remote application requesting a socket call.

Although the TRSS Parent does not access the MLS LAN, its child processes do. Hence, the TRSS Parent must run at the level of the MLS LAN. However, it must also be able to receive signals from remote applications running at the level of the user who invoked them. Furthermore, it must be able to switch to the user ID of the user who requested the remote application invocation before it initializes the MSKT Database, so that the remote application has read/write access to the database. For these reasons, the TRSS Parent is designated a trusted process.

i. Trusted Remote Session Server (TRSS) Child

The TRSS Child process is responsible for handling socket calls on behalf of its assigned remote application. The two processes communicate via a MYSEA socket (MSKT) allocated by the TPS Child process. Any time the RA requires access to the MLS interface, it enters its socket request into the MSKT Database and (for all but the first call) signals the TRSS Child process. The TRSS Child process then retrieves the requested call type and corresponding parameters from the MSKT Database and makes the appropriate socket call on behalf of the RA. Once the function call has completed,

the TRSS Child sets the return value of the call and any data received into the appropriate fields in the database entry, and signals the RA process. The RA is then able to retrieve the results of its socket call from the database.

When an RA process requests access to a peer (via the *connect, sendto, accept,* or *recvfrom* socket calls), the TRSS Child process is responsible for checking whether the connection is allowed before proceeding with the call. It does this by consulting the Peer Level Database and the User Database to retrieve the current level of the peer. By comparing the session level of the peer with the session level of the RA, it is able to determine whether the access is allowable. If so, the TRSS Child must consult the Source Address Binding Database to determine the source address configured for use with the supplied destination address before binding the socket to a source port/address pair.

For each new connection established on behalf of the RA process, the TRSS Child updates the Remote Connections Database with the new connection and its current session level. Upon the closing of each connection, the TRSS Child removes the connection from the database.

The TRSS Child runs at the level of the MLS LAN. Because it requires access to privileged ports, and because it must also communicate with the RA running at the session level of the user who invoked it, the TRSS Child is designated a trusted process.

j. Remote Application (RA)

The remote application process is an application process executing on the server at the request of a remote client. The RA process is untrusted, and runs at the session level of the user who invoked it. The RA may only request socket operations on the MLS LAN by way of an MSKT managed by a TRSS Child process, as described above. If an RA is ported from another platform, its socket calls must therefore be modified to use the MSKT interface.

Table 1 presents a summary of the processes involved in the execution of remote applications.

Process	Trusted?	Session Level	Launched By	Instances
TPS Parent	Yes	MLS LAN	(System Daemon)	One/System
TPS Child	Yes	MLS LAN	TPS Parent	One/TPE
SSD	Yes	MLS LAN	(System Daemon)	One/System
SSS Parent	Yes	MLS LAN	SSD	One/Allowed Protocol
SSS Child	Yes	MLS LAN	SSS Parent	One/APS-Client Connection
APS (*httpd*)	No	Remote User or RA	SSS Child	One/SSS Child
CGI RA Invocation process	No	Remote User	APS (*httpd*)	One/RA Invocation Interface Request
TRSS Parent	Yes	MLS LAN	TPS Child	One/User-Session-Level
TRSS Child	Yes	MLS LAN	TRSS Parent	One/Remote App.
Remote App.	No	Remote User	CGI RA Invocation process	One/RA Invocation Request

Table 1. Process Attributes

Table 2 summarizes the privileges required by each trusted process.

Trusted Process	Privileges Required
TPS Parent	• Ability to access the MLS LAN as well as system-low databases.
TPS Child	• Ability to access the MLS LAN as well as system-low databases. • Ability to access user identification and authentication information.
SSD	• Ability to fork and terminate child processes that run as the *network* user.

Trusted Process	Privileges Required
SSS Parent	• Ability to run as the *network* user in order to access privileged ports. • Ability to fork children that can access the MLS LAN and also communicate with user-level APS processes. • Ability to fork children that can switch user IDs.
SSS Child	• Ability to access the MLS LAN and also communicate with user-level APS processes. • Ability to switch to the user ID of the remote user before initializing the MSKT Database and launching the APS process.
TRSS Parent	• Ability to switch to the user ID of the remote user before initializing the MSKT Database. • Ability to fork child processes with access to the MLS LAN and also communicate with user-level RA processes. • Ability to fork and terminate child processes that run as the *network* user. • Ability to be launched by low-integrity processes. (See Chapter IV for a discussion of the *Trusted Parent Exempt* privilege.)
TRSS Child	• Ability to access the MLS LAN and also communicate with user-level RA processes. • Ability to run as the *network* user in order to access privileged ports.

Table 2. Privilege Requirements of Trusted Processes

3. Databases

a. Allowed Protocols Database

The Allowed Protocols Database is a static, per-system database that lists the application protocols provided by the MYSEA server. Each entry, corresponding to a single allowed protocol, consists of a descriptive identifier of the protocol (e.g., "HTTP"), a path to the executable that will provide the protocol service, and the port number on which it will listen for requests.

The SSD process uses the Allowed Protocols Database to start an SSS Parent process for each allowed protocol.

b. Allowed TPE Database

The Allowed TPE (Trusted Path Extension) Database is a static, per-system database that lists the TPEs authorized to log in to the MYSEA server. Each entry consists of the IP address of an allowed TPE.

The TPS Parent process uses the Allowed TPE Database to restrict login access to authorized clients.

c. User Database

The User Database is a per-system database that is used to associate communications from a particular TPE with a specific user and session level. There is one entry in the database for each active TPE.

The database is maintained by the TPS Parent and Child processes, which update it every time a user logs in, changes his/her session level, or logs out. The SSS Parent and Child processes use the database to determine the proper session level to associate with APS processes launched to handle requests from TPEs. The SSS Child and TRSS Child processes also make use of the database when an APS or RA requests a connection with a peer, and that peer is not contained in the Peer Level Database. In this case, it is assumed that the APS or RA process is attempting to communicate with a TPE; hence, the SSS or TRSS Child process must check to see that this TPE is logged in at a session level acceptable for communication with the requesting APS or RA.

d. Remote Connection Database

The Remote Connection Database is a per-system database that is used to bind connections initiated by RAs and APSs to specific users and session levels. It is initialized by the TPS Parent, and updated by the TRSS Child or SSS Child each time the RA or APS requests the establishment or termination of a remote connection. Each remote connection entry consists of a source port, a source IP address, a destination port, and a destination IP address.

The SSS Parent uses the database to check whether incoming traffic is the result of a connection request by an RA or APS; it does this only if it fails to find a record

of the TPE in the User Database. If the SSS Parent locates the remote connection in the database, the incoming data is passed on to the appropriate SSS Child process; otherwise, the connection is dropped.

e. Remote Application MYSEA Socket Map (RAMSKT Map) Database

The RAMSKT Map Database is a per-system database that is used to map a particular user and session level to the MYSEA Socket (MSKT) Database and TRSS Parent process assigned to provide remote application support for that user/session-level pair. The TPS Parent initializes the memory block that contains the database, and the TPS Child allocates an entry for each authenticated user. The database is read by the TRSS Parent and Child processes and by the RA to look up the MSKT Database handle to be used for socket communications between them.

f. Application Protocol Server MYSEA Socket Map (APSMSKT Map) Database

The APSMSKT Map Database is a per-system database that is used to map a particular user and session level to the MYSEA Socket (MSKT) Database assigned to provide APS support for that user/session-level pair. The TPS Parent initializes the memory block that contains the database, and the TPS Child allocates an entry for each authenticated user. The database is read by the SSS Child and APS processes to look up the MSKT Database handle to be used for socket communications between them.

g. MYSEA Socket (MSKT) Database

The MSKT Database is a structure used to pass data between TRSS and remote application processes, and between SSS Child and application protocol server processes. Two instances of the database are initialized for each active user/session-level combination: one instance for RA support, and the other for APS support. Because the untrusted RA and APS processes require read and write access to their respective

databases, each database must be initialized at the security level of the untrusted process (i.e., at the session level of the remote user or remote application that requested the RA or APS invocation).

Each entry in the MSKT Database contains the following fields:

- Call Type: This field is used to indicate to the TRSS Parent or Child process what type of socket call is being requested by the remote application, or to indicate to the SSS Child process the type of socket call being requested by the application protocol server. The supported socket calls are listed later in this section.
- Data Buffer: This field is used to pass data between the TRSS Child and the RA, or between the SSS Child and the APS. It is used for the *read, recv, recvfrom, write, send, sendto, getsockopt,* and *setsockopt* functions.
- Function Return Value: This field is used by the TRSS Child process to pass the return value of the function call to the RA, or by the SSS Child process to pass the return value of the function call to the APS.
- MSKT FD: This field contains the socket file descriptor associated with the MSKT.
- Untrusted PID: This field contains the process ID of the untrusted process (i.e., the RA or APS) accessing the MSKT.
- Trusted PID: This field contains the process ID of the trusted process (i.e., the TRSS or SSS Child) accessing the MSKT.
- Other fields include an 'In Use' flag, an Errno field, a Parameter Buffer (only used by certain functions), and a Variable Buffer Flag that indicates whether the Data Buffer or Variable Shared Memory Buffer is being used to pass data.

The MSKT Module currently supports the following socket library function calls: *accept, bind, close, connect, fcntl* (to a limited extent), *fork*, *getpeername*, *getsockname*, *getsockopt, ioctl* (to a limited extent), *listen, read, recv, recvfrom*, *select* (to a limited extent), *setsockopt, send, sendto, shutdown, socket*, and *write*.

For more detail on the implementation and current limitations of these function calls, see [3].

h. Peer Level Database

The Peer Level Database is a static, per-system database that is used to indicate whether a peer is multilevel, and if not, to associate the peer IP address with a specific security level.

The database is used by TRSS Child and SSS Child processes to determine whether a connection requested by a RA or APS with a specific remote peer is allowable.

i. Source Address Binding Database

The Source Address Binding Database is a static, per-system database that is used to associate one or more destination IP addresses with a source IP address. The need for this database arises from the fact that a remote connection must be registered in the Remote Connection Database in order for the SSS Parent to recognize it and handle incoming traffic for the connection appropriately. In order to register a connection in the database, there must be known values for each of the following fields: source port, source IP address, destination port, and destination IP address. This is problematic for certain socket calls, such as *connect*, that do not necessarily bind to a source IP address before attempting to establish a remote connection.

To account for this, the TRSS Child process making the call on behalf of a remote application may consult the Source Address Binding Database to determine the appropriate source address to be used with the given destination address. The database consists of pre-configured source IP addresses and their corresponding network masks and destination IP addresses. The Source Address Binding Module provides an interface

for determining the correct source address for a given destination address. This function iterates through each entry in its database, performing a bit-wise *AND* of the input destination address with the current entry's network mask, and comparing the result with the masked destination address for that entry. If there is a match, the function returns the source address associated with the entry.

j. Cleanup Database[2]

The Cleanup Database is a per-system database that maintains a list of the SSS Child processes that need to be terminated when a user changes session levels or logs out. Each database entry contains two fields: an SSS Child process ID, and the associated TPE ID. The database is initialized by the TPS Parent process; SSS Child processes enter themselves into the database as they are launched and remove themselves before they terminate. Because they are trusted, inadvertent corruption of the database by SSS Child processes is not a concern.

The TPS Child process refers to the Cleanup Database whenever a user logs out or changes session level. It uses the TPE identifier to retrieve the process ID of each active SSS Child process associated with that TPE, and terminates those processes.

Table 3 summarizes the initialization properties of each database.

Database	Initializing Process	Frequency	Security Level	Statically Configured	Shared Memory
Allowed Protocols	SSD	Per-system	MLS LAN	Yes	No
Allowed TPE	TPS Parent	Per-system	MLS LAN	Yes	No
User	TPS Parent	Per-system	MLS LAN	No	Yes
Remote Connection	TPS Parent	Per-system	MLS LAN	No	Yes
RAMSKT Map	TPS Parent	Per-system	System-low	No	Yes
APSMSKT Map	TPS Parent	Per-system	System-low	No	Yes

[2] The Cleanup Database is currently undergoing re-design, and is not fully implemented at this time. Cleanup-related issues are discussed further in the "Future Work" section of Chapter VI.

Database	Initializing Process	Frequency	Security Level	Statically Configured	Shared Memory
MSKT	SSS Child and TRSS Parent	Two per TPE user/session-level (one for APSs and one for RAs)	Session level of requesting user or RA	No	Yes
Peer Level	TPS Parent	Per-system	MLS LAN	Yes	Yes
Source Addr. Binding	TPS Parent	Per-system	MLS LAN	Yes	Yes
Cleanup	TPS Parent	Per-system	MLS LAN	No	Yes

Table 3. Database Initialization Summary (after [3])

Table 4 summarizes the database access privileges required by each process. An 'R' signifies that a process requires *read* access to the database; a 'W' signifies that the process requires *write* access to the database. An asterisk indicates that the process is untrusted.

Process \ Database	Allowed Protocols	Allowed TPE	User	Remote Connection	RAMSKT Map	APSMSKT Map	MSKT	Peer Level	Source Addr. Binding	Cleanup
TPS Parent		R W	R W	R W	R W	R W		R W	R W	R W
TPS Child			R W		R W	R W				R
SSD	R W									
SSS Parent			R	R						
SSS Child						R	RW	R	R	R W
APS*						R	RW			

Process \ Database	Allowed Protocols	Allowed TPE	User	Remote Connection	RAMSKT Map	APSMSKT Map	MSKT	Peer Level	Source Addr. Binding	Cleanup
TRSS Parent					R		RW			
TRSS Child			R	R W			RW	R	R	
Remote App.*					R		RW			

Table 4. Process-Database Relations

4. Other Modules

a. Semaphore

The Semaphore Module provides an interface for locking access to databases. Databases are locked whenever necessary to prevent the occurrence of race conditions, i.e., the reading of a database by one process while a second process is busy writing to that database, which can lead to inconsistencies in the data presented to the reading process. The MSKT, RAMSKT Map, APSMSKT Map, Remote Connection, and Cleanup Databases depend on this module.

b. User Identification and Authentication

The User Identification and Authentication (I&A) Module manages access to the XTS's trusted services for user identification and authentication. It is used by the TPS Child to check the validity of username/password and username/session-level combinations provided by remote users attempting to log in or set their session levels.

c. Privileges

The Privileges Module manages access to the XTS's trusted services for granting and revoking privileges. These privileges can enable the invoking process to do one or more of the following:

- Bypass MAC and/or DAC controls.
- Perform Identification & Authentication (I&A) checking.
- Change the owner or group attributes of the current process.
- Change the access class of an object.
- Upgrade the mandatory access level of an object.
- Read objects with a lower integrity.

The module also provides the means to restore the invoking process's previous set of privileges. Only trusted applications that have been authorized for specific privileges in advance may take advantage of the functions offered by this module. The SSD, SSS Parent and Child, TPS Parent and Child, and TRSS Parent and Child processes all make use of the module.

d. Shared Memory

The Shared Memory Module provides an intuitive interface for accessing shared memory on the XTS-400 system. This module is used to facilitate the sharing of databases between processes.

e. Utility

The Utility Module provides an intuitive interface for several of the utility functions provided by the STOP OS, including the means to display and compare session levels, and various debugging functions.

f. MYSEA Synchronization

The MYSEA Synchronization Module implements an intuitive interface for synchronization functions provided by the STOP OS. Specifically, it provides the means for processes to signal each other and pause until signaled using the IPC message-passing mechanism. These functions are utilized in communications between the SSS and APS processes, and between the TRSS and RA processes.

g. Socket Handler

The Socket Handler Module presents the interfaces for the socket handler functions used by the TRSS and SSS Child processes to perform actual socket calls. The results of the socket calls are passed back to the APS or RA by way of the MSKT Database.

D. PRE-IMPLEMENTATION MODIFICATIONS TO DESIGN

The design concepts discussed in this chapter were developed in a previous thesis [3], with a couple of exceptions. Subsequent to the publication of the thesis, certain inefficiencies were identified in the use of the pseudo-socket layer and in the named-pipe method for inter-process communication. These inefficiencies, and the design changes that resulted, are described in the following sections.

1. MYSEA Sockets (MSKTs)

The original design for remote application support called for the creation of a Remote Application Pseudo-Socket (RAPSKT) Database to be used by the TRSS and RA processes. This database was to be distinct from the Pseudo-Socket (PSKT) Database already in use by the SSS and APS processes for the purpose of handling application protocol requests. During the course of testing, it was noted that the performance of the application protocol servers degraded severely as the number of concurrent application protocol requests increased. Rather than try to debug the pre-existing PSKT code, it was decided to use the RAPSKT interface to handle socket operations for both the application protocol servers and the remote applications. This was possible because the two types of

databases served similar purposes: to pass socket requests and data between untrusted processes requiring access to the MLS LAN and the trusted processes authorized to make socket library calls on their behalf. The database referred to throughout this document as the MSKT (MYSEA Socket) Database is equivalent to the RAPSKT Database described in [3], revised slightly to accommodate the requirements of the application protocol servers in addition to those of remote applications.

2. Inter-Process Signaling

The method for inter-process signaling between the RA and TRSS processes has also been revised since the publication of [3]. The TRSS processes can signal the RA process using the standard signaling mechanism, since the TRSS Parent and Child are trusted processes with the privilege to communicate with processes running at different levels. However, the untrusted RA process cannot signal back to any process running at a different level than itself, requiring the introduction of an alternative signaling mechanism.

The original design called for the RA process to signal the TRSS Parent and Child processes by way of *named pipes*. This involved the creation of a named pipe by the TRSS process with a file name ending in its own process ID. The TRSS process would wait for a signal by the RA by calling select on the file descriptor of the named pipe and waiting for the RA to write to the file. When the RA first needed to communicate with the TRSS Parent or Child process, it would look up the TRSS process ID in the RAPSKT Database and write to the corresponding named pipe. The TRSS process would then read the data from the file descriptor, respond to the RA using the standard signaling mechanism, and again call *select* on the file descriptor to wait for the RA's next signal. The advantage of this method was that it seemed well-suited for the "lock-step" style of communication between the RA and TRSS processes: the RA is required to signal the TRSS Parent when it requires a new socket connection, and pause for the parent to respond with the results of the call; once the TRSS parent responds, the RA resumes processing and eventually signals the TRSS Child to handle the next socket operation, pausing for its response; the TRSS Child returns a value, and pauses for the RA's next

signal; and the cycle repeats for each socket operation required by the RA thereafter. The named-pipe method is well-suited to handle this type of communication pattern, but it does have disadvantages:

- It cannot be generalized to handle more complex communication models, possibly involving more than two processes or more than two states (currently *sleep* and *awake*);
- It carries with it the inconvenience of having to create the named pipe and a deflection directory [13] in which to store it.

For these reasons, it was decided to instead make use of the XTS-400 proprietary IPC (Inter-Process Communication) mechanism, or TCB messaging, packaged in an intuitive interface within the MYSEA Synchronization Module. This method provides all the functionality of the named-pipe method, with the advantage that it is directly supported by the operating system and doesn't require the special creation of named pipes or deflection directories.

This change also applies to inter-process signaling between the SSS Child and APS.

E. OVERVIEW OF IMPLEMENTATION REQUIREMENTS

1. Newly Implemented Components

a. Processes

The following processes were implemented specifically for the purpose of remote application support:

- TRSS Parent Process
- TRSS Child Process
- Remote Applications (tftp [14] and swget clients)
- CGI Remote Application Invocation Scripts

These processes were implemented as part of this thesis.

b. Databases

Additionally, the following databases were implemented specifically for the purpose of remote application support:

- Remote Connection Database
- RAMSKT Map Database
- MSKT Database
- Peer Level Database
- Source Address Binding Database
- Cleanup Database

These six databases and their interfaces were implemented prior to the start of this thesis.

c. Modules

Finally, the following modules were implemented for the purpose of remote application support prior to the start of this thesis:

- MYSEA Synchronization Module
- Socket Handler Module

2. Modifications to Existing Components

Many of the components that are affected by the implementation of remote application support were previously designed and implemented [15], and existed for their own purposes outside the scope of remote application support. These included the SSD, SSS, and TPS processes, as well as the Allowed Protocols, Allowed TPE, and User Databases. The majority of the modules described in Section C4 (all but the MYSEA Synchronization and Socket Handler Modules), also existed prior to this project. However, several of these pre-existing components underwent revision during the course of the project. This included the adaptation of the SSS and APS processes to make use of the MSKT Database, as well as other minor revisions to the SSD, SSS Parent and Child,

and TPS Parent and Child processes, and to the User Database. These revisions were largely aimed at improving cleanup processing, and are described in greater detail in [3].

F. SUMMARY

The design specifications summarized in this chapter provide a complete framework for remote application support within the MYSEA environment. An implementation based upon these requirements was completed for this project. The implementation process, including module-specific implementation requirements and unforeseen issues encountered during development, are described in Chapter IV.

IV. IMPLEMENTATION

A. OVERVIEW

This chapter describes the implementation and modification of the remote application support components developed as part of this thesis. These include modifications to the TPS Child process, the implementation of the TRSS Parent and Child processes, the development of two proof-of-concept remote applications (a TFTP client ported from publicly available source code, and a simple web client), and the implementation of two remote application invocation mechanisms in the form of CGI scripts. This chapter also describes unforeseen issues that arose during the course of implementation, and other deviations from the specifications in [3].

Within each of the following sections, a reproduction of the Process Overview figure (Figure 1) is presented as a reminder of the context in which the current RA component of interest was designed to function. Lightly shaded boxes represent processes implemented during this project; darkly shaded boxes designate the components currently under discussion.

Appendix A provides a listing of the MYSEA source code files that were created or modified for this project.

B. DEVELOPMENT ENVIRONMENT

The MYSEA remote application support code was developed on an XTS-400 system running the STOP 6 operating system. The MYSEA software, including the TPS and TRSS processes, was implemented in C and compiled using gcc 3.2, as were the proof-of-concept remote applications. The CGI scripts were written in Perl and executed using Version 5.8.0 of the Perl interpreter. Both the C compiler and Perl interpreter were the standard versions shipped with the XTS-400 system. Remote application support was integrated into Version 1.1x of the MYSEA server software.

C. IMPLEMENTATION DETAILS

1. TPS Child Process

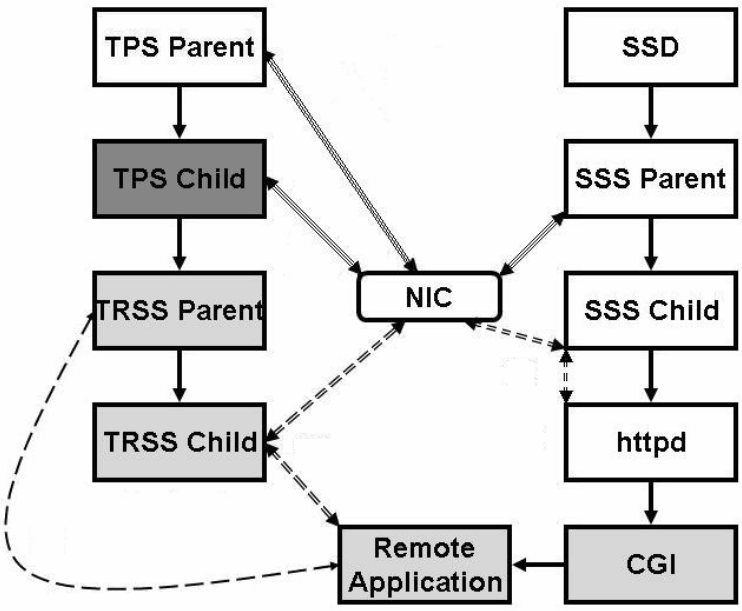

Figure 2. TPS Child in the Context of Remote Application Support

The TPS Child process was implemented long before this project, but required the addition of new functionality in order to set the stage for remote application support for the particular TPE for which it was invoked. Specifically, it was modified to take the following actions upon the successful establishment of a user session:

1. Create a TRSS Parent process to handle socket requests from remote applications launched during the session;
2. Allocate shared memory to be used for the MSKT Database for the session, and store the handle to the MSKT Database in the RAMSKT Map Database entry associated with the session; and
3. Set the process ID of the newly created TRSS Parent process in the RAMSKT Map Database entry associated with the session.

During the course of implementation, it was clarified that the successful establishment of a user session meant the invocation of the “run” command by the user

after having successfully logged in and negotiated a session level. The specifications in [3] called for the execution of the above steps immediately following a successful user login; however, at this point, the user is still operating in the context of the Trusted Path and may elect to change his or her session level before running an untrusted session. Because the steps listed above depend on the previous determination of the user/session-level pair for which the remote application support structures and processes should be initialized, they should be executed after the session level negotiations have been completed and the user exits the Trusted Path session. Furthermore, the user may wish to re-invoke a Trusted Path session and re-negotiate the session level at a later point within the same login session. In this case, the remote application support mechanism will only be functional for the re-negotiated session if it is re-initialized each time the user issues the "run" command after changing session levels, rather than being initialized only once after the user's initial login.

Before writing to the RAMSKT Map Database in Steps 2 and 3, the TPS Child must enable its MAC/DAC exemption privilege. This is necessary because the TPS Child runs at the level of the MLS LAN, whereas the RAMSKT Map Database is a system-low database. (The TPS Parent must similarly enable its MAC/DAC exemption privilege before initializing the RAMSKT Map Database.) These requirements were identified subsequent to the publication of [3].

2. TRSS Parent and Child Processes

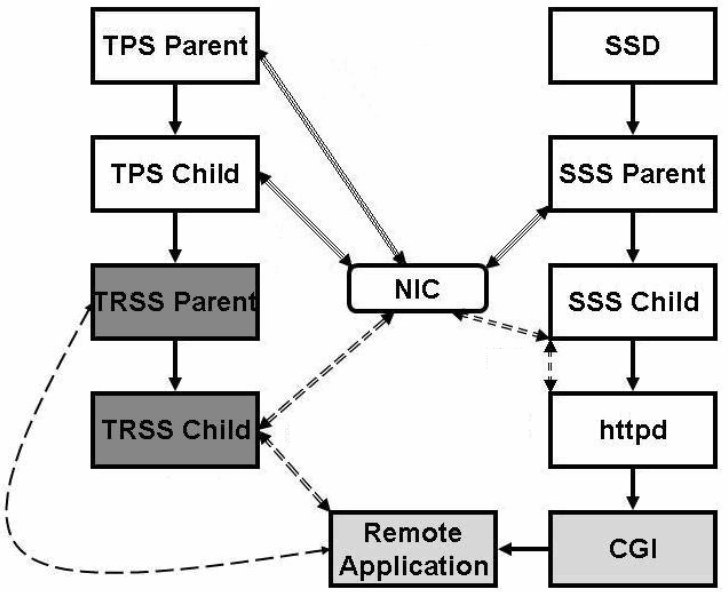

Figure 3. TRSS Processes in the Context of Remote Application Support

The TRSS Parent and Child processes, highlighted in Figure 3, were implemented specifically for the purpose of this project, per the specifications in [3]. In the interest of efficiency, a few deviations were made from the specifications:

1. The TRSS Parent process does not loop while waiting for its MSKT Database handle to be successfully retrieved from the RAMSKT Map Database; instead, it sleeps until signaled by the TPS Child process that invoked it, indicating that the handle is ready to be retrieved.
2. The TRSS Child process does not retrieve the process ID of the remote application to be serviced by performing an MSKT Database lookup; instead, it inherits the process ID from the TRSS Parent that forked it. (The TRSS Parent is signaled by the remote application requiring service and records the process ID of the signaling process before it forks the TRSS Child.)
3. The TRSS Child process does not retrieve the handle for the MSKT Database entry to be attended by doing a linear search for its own process ID in the MSKT Database; instead, it inherits the handle from the TRSS Parent that

forked it. (The TRSS Parent locates the database entry containing a new *socket* call prior to forking the TRSS Child.)

Furthermore, the following deviation from specifications was required for the proper functioning of remote applications:

- The TRSS Parent, which normally runs as the *network* user, must switch to the user ID of the remotely authenticated user before initializing the MSKT Database. This is because the MSKT Database is configured to be readable and writable only by its owner (i.e., by processes running with the same user ID as the process that initialized it), and untrusted remote applications running with the user ID of the remote user who invoked them must be able to access the database.

The TRSS binary must be designated a trusted program with the specific privileges necessary for it to perform its duties. This requires using the trusted *tp_edit* program provided by the STOP 6 operating system to designate the TRSS program as trusted, and to grant it the privileges it requires. These include MAC/DAC exemption and the ability to set its user ID, as documented in Table 2. Installation procedures for the TRSS program are provided in Appendix B.

3. Remote Applications

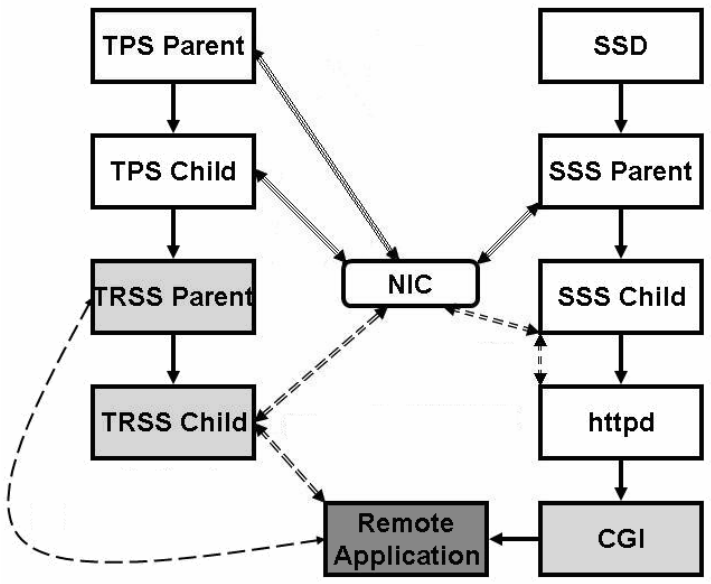

Figure 4. TFTP Client and Swget in the Context of Remote Application Support

To demonstrate the functionality of the newly implemented remote application support mechanism, two remote applications were installed on the MYSEA server: a Trivial File Transfer Protocol (TFTP) client ported from publicly available source code, and a simple web client (swget) developed specifically for use as a remote application on the MYSEA server. The remote application component is illustrated in Figure 4.

a. Trivial File Transfer Protocol (TFTP) Client

As one means of demonstrating the functionality of remote network-enabled applications, a TFTP client was ported onto the MYSEA server and adapted to make use of the MYSEA socket (MSKT) mechanism. This was accomplished by making the remote application modifications deemed necessary in [3]. Specifically, the TFTP client source code was modified to do the following:

- Locate and attach to the MSKT Database that the client would be using for socket requests. This required that the client take the following sequence of steps:
    1. Initialize access to the RAMSKT Map Database;

2. Retrieve the MSKT Database handle to be used by the remote application from the RAMSKT Map Database; and
3. Initialize access to the MSKT Database.

These steps had to be taken before any socket operations were attempted.

- Use MSKT socket calls instead of their conventional equivalents (e.g., *mskt_socket* instead of *socket*, *mskt_recvfrom* instead of *recvfrom*).

  Successful execution of the TFTP GET command required use of the *mskt_socket*, *mskt_bind*, *mskt_sendto*, and *mskt_recvfrom* functions.

- Detach from the RAMSKT Map and MSKT Databases when signaled by the TRSS Child process to terminate.

  This required the initialization of a *sigaction* structure and the definition of a signal handler function that detached from the RAMSKT Map and MSKT Databases before exiting. The *sigaction* system call was used to specify that the process should respond to a SIGTERM (a signal to terminate) by calling its customized signal handler function.

In order to integrate the TFTP client with the CGI invocation interface, additional client modifications were required. Traditionally, TFTP has operated only in interactive mode; i.e., the user must enter separate commands to invoke the client (`tftp`), transfer files (`get <remote_file_name> <local_file_name>`), and terminate the client (`quit`). For MYSEA RA testing purposes, it was necessary for the entire TFTP client invocation, file transfer, and client termination to be automated so that the transfer could be requested, and the results captured, by the execution of a single external command by a CGI script. Using the tftp-0.41-hpa client (an enhanced version of the original BSD TFTP client) source code [14] as a baseline, only a few lines of code needed to be added

to main.c in order to implement a single new command (`getq`) that would retrieve a file and terminate the client process in sequence. The `getq` command and its arguments could then be piped to the TFTP client application on the command line as shown:

```
echo getq <remote_file_name> <local_file_name> | tftp <server_name>
```

where the remote file name, local file name, and server name were all provided as user inputs to the CGI script. The CGI script only needed to issue this single command to launch the TFTP client, download the file requested by the user, and place it in the directory requested by the user, meanwhile capturing the report of the transfer to be presented back to the user. A successful invocation of the `tftp getq` command specified above results in output of the form:

```
tftp> Received X bytes in Y seconds [Z bit/s]
```

b. Web Client

To demonstrate the functionality of remote applications interacting with a local web (APS) server, a simple web client was implemented for use on the MYSEA server. The client was modeled after the text-based wget client widely used on Linux and UNIX platforms, but was designed to make use of MSKT socket calls. Because it is much simpler than the popular wget client, the client implemented for the purpose of this project was named *Simple wget*, or *swget*.

The swget client takes, as its single command-line argument, the URL of the web page to be retrieved, and parses the URL to determine the name of the server to be contacted and the name of the file to be retrieved from the server. It then issues an "HTTP GET" command to port 80 of the requested server for the desired file name, stores the response from the web server, and filters the HTTP header information from the data received before printing it to STDOUT. The result displayed to the user is the HTML markup content of the requested web page.

The swget client was developed for use as a proof-of-concept remote application. Its practical use is limited, as it issues only a single HTTP request for the file specifically requested by the user. Should that file include links to embedded components, such as images or sounds, those components will not be retrieved. A port of a fully fledged web client such as wget is left for potential future work.

4. CGI Remote Application Invocation Processes

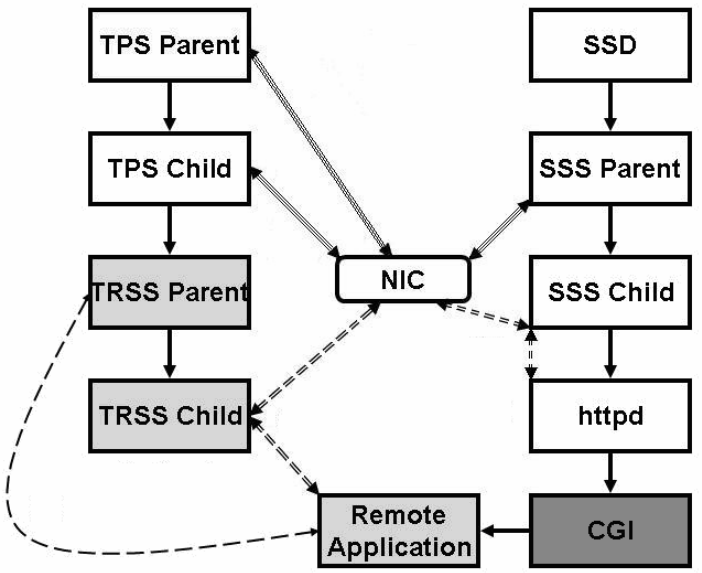

Figure 5. CGI RA Invocation Processes in the Context of Remote Application Support

Two distinct CGI interfaces were developed during this project as a means for invoking remote applications. Each was designed to be platform-neutral, and may be accessed via a web browser on the MYSEA client. The first interface takes the form of a command menu, from which the user may select the desired command (i.e., remote application) to be executed on the MYSEA server. The second takes the form of a web shell, through which the user may enter UNIX-style commands to be executed on the server. In each case, the output of the requested command is embedded in a web page served back to the user. The CGI component is highlighted in Figure 5.

a. Command Menu

The command menu includes the options to perform simple tasks such as moving and copying files, changing directories, listing directory contents, and displaying the contents of text files. It also includes the option to launch the TFTP or web client on the MYSEA server in order to request the transfer of a file onto the server. A screen shot of the command menu is presented in Figure 6.

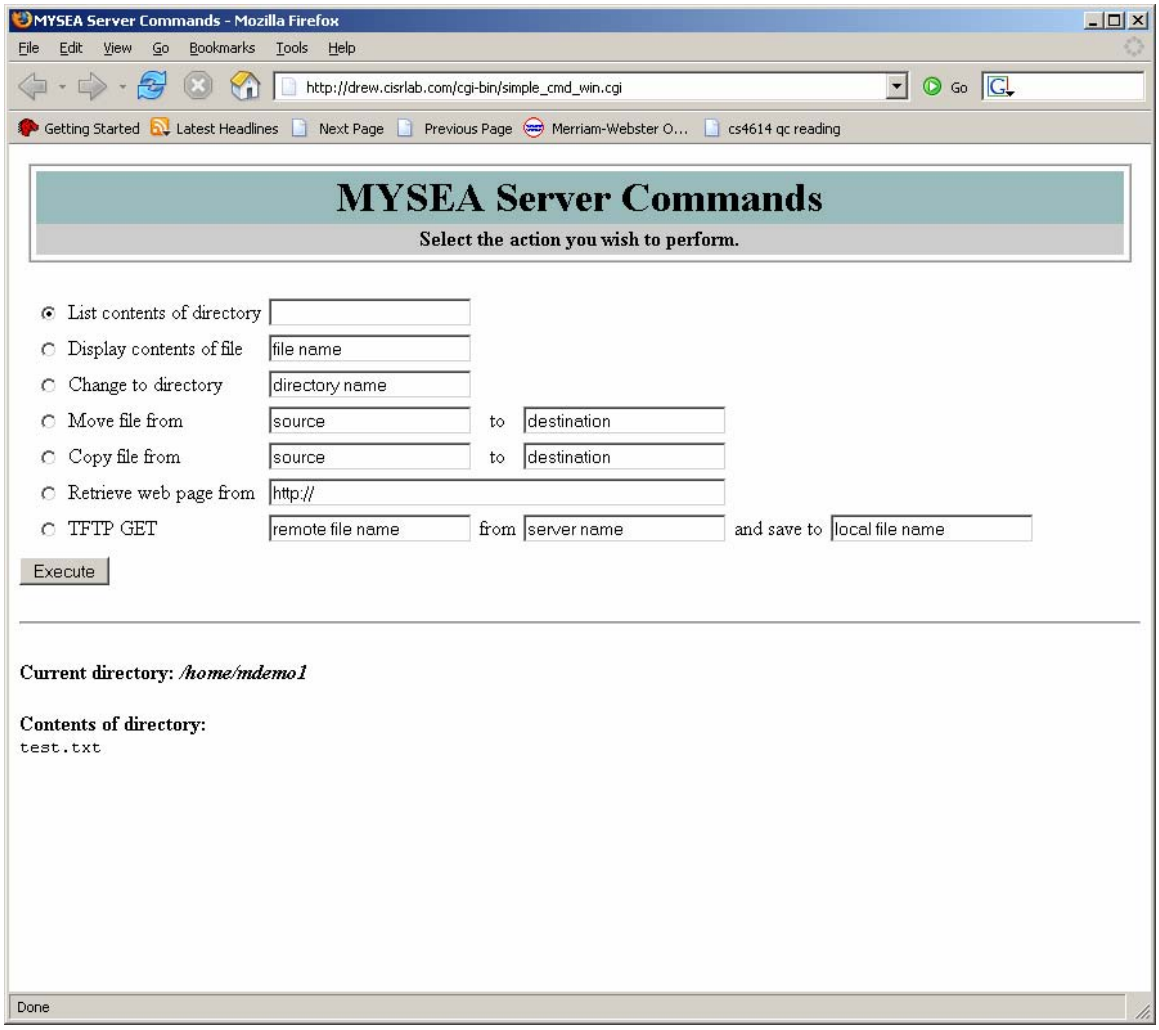

Figure 6. Command Menu

User commands entered via the command menu are interpreted, and the corresponding UNIX commands executed on the MYSEA server, by a CGI script running on the server. For example, a user may request a listing of files in the directory

test_dir by selecting the "List contents of directory" option from the command menu and typing the directory name *test_dir* in the corresponding text field. The CGI script will then run the `ls test_dir` command on the server and report the results back to the user.

The command menu interface has been implemented as a Perl script. The major functions of the script are to:

- Generate the initial command menu as an HTML form document;
- Parse the command selections and arguments entered and submitted by the user via HTTP POST;
- Perform sanitization and error-checking of user input;
- Transform the user input into valid UNIX commands (`ls, mv, cp, cd, tftp, or swget`) with the appropriate command-line arguments;
- Execute those commands on the server;
- Capture any output generated by the commands on STDOUT or STDERR; and
- Upon completion of each command, embed the captured output into an HTML page and present it back to the user. In the case of a web page request, the requested page will replace the original command menu. In all other cases, the resulting web page will contain the original command menu followed by a report of the results of the user's most recently executed command.

The "Change Directory" command required special handling. Because the corresponding UNIX command, `cd`, is not an actual binary to be executed but rather a built-in function of popular command-line shells (e.g., bash and csh) designed to influence the subsequent behavior of the shell, its functionality had to be implemented within the CGI script. This was accomplished by calling Perl's internal *chdir* function with the user-specified directory as an argument. This allows the user to subsequently refer to files and directories by their relative paths from the current working directory rather than their absolute paths. However, because the CGI scripts are inherently stateless (i.e., an entirely new CGI process is invoked for each command request

submitted by the user), the proper implementation of the `cd` command required the introduction of some mechanism to transfer the record of a previously requested current working directory across multiple invocations of the script. To this end, a hidden form field containing the name of the current working directory was embedded within the command menu presented to the user after the completion of each command request. Whenever the script changes its current working directory, it also updates the value of this hidden form field. When the user next submits the form, the CGI script retrieves the name of the user's current working directory from the hidden field and calls *chdir* with the retrieved value as an argument before handling the user's request. If the user has not yet requested a `cd` operation, the current working directory is set to the user's home directory, as specified in the "/etc/passwd" file on the MYSEA server. If the user has been assigned no home directory, the current working directory defaults to the parent directory of the CGI script ("/home/http/cgi-bin/").

b. Web Shell

The web shell was implemented based upon similar principles, but rather than requiring the user to select between pre-determined command options, it allows the user to enter arbitrary UNIX-style commands into a text field whose contents are interpreted as commands to be executed on the MYSEA server. The output from the command execution is embedded into an emulated terminal window presented back to the user.

Original plans called for the port of a pre-existing web shell such as the one made publicly available by the Gamma Group [16]. However, because the standard STOP 6 development environment lacked certain Perl modules required by the Gamma shell (namely, the Perl CGI Module, which automates certain CGI tasks), it was decided to implement a simple web shell from scratch. The MYSEA web shell was implemented almost identically to the command menu, with the exception that user-requested commands are not checked against a list of valid commands, but are instead executed directly on the MYSEA server without intermediate interpretation.

A screen shot of the web shell is presented in Figure 7.

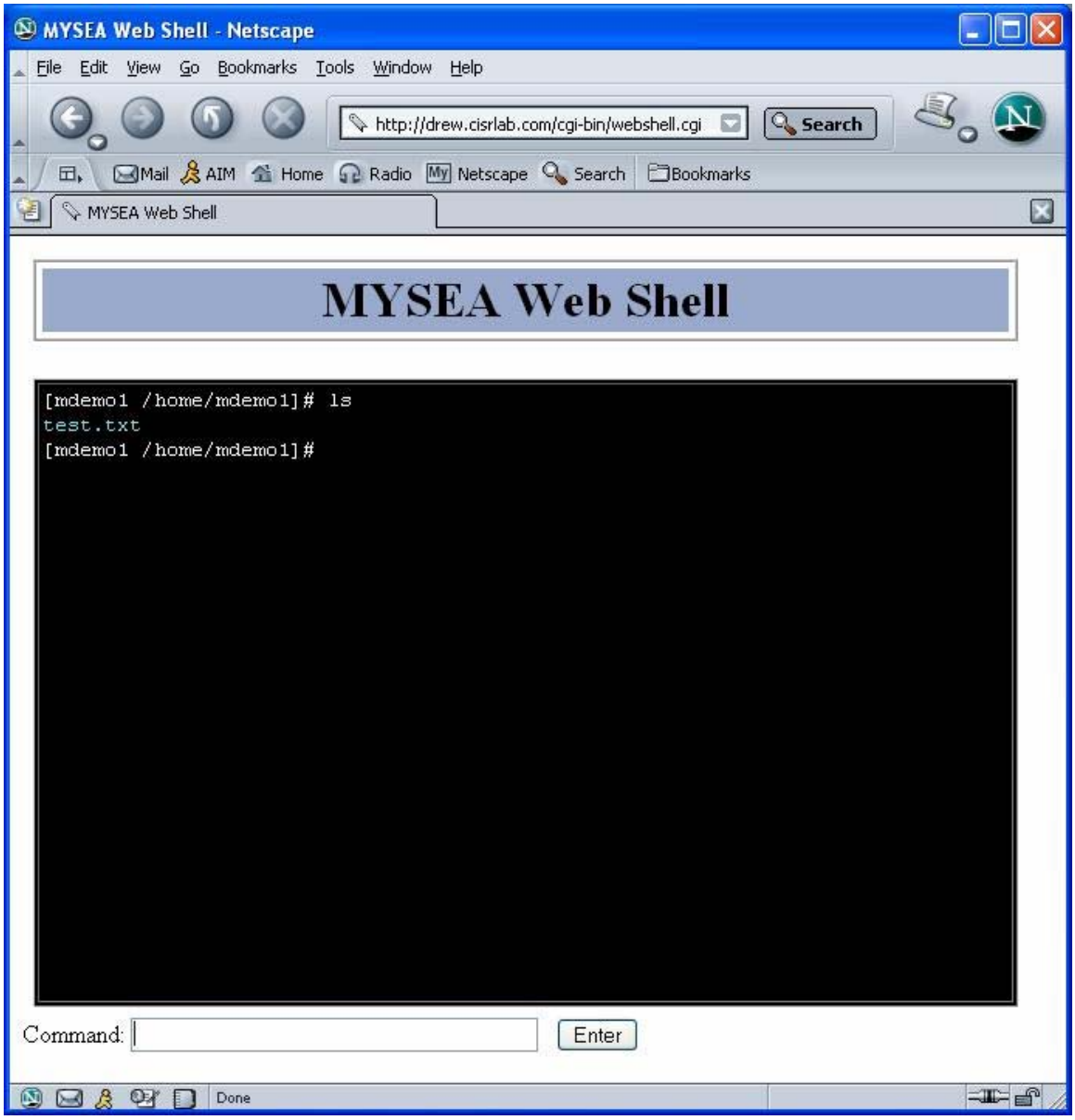

Figure 7. Web Shell

Although the web shell provides basic remote command-line functionality, it has significant limitations. Most notably, any command entered by the user must terminate before the web shell will display its output. This precludes the execution of interactive applications that prompt the user for input before displaying output or terminating. (In some cases, a user may work around this restriction by making use of pipes, as in the TFTP client invocation method presented earlier in this chapter.) The

reason for the limitation lies in the method used by the web shell to interact with remote applications. The web shell CGI script makes use of Perl's "backtick" method of spawning external programs and capturing their output; using this method, the CGI process must wait for its spawned child process to terminate before continuing its own processing. For example, a Perl script containing the following line would execute 'ls' as a child process, wait for it to terminate, and copy its output from STDOUT to its own local $output variable:

```
$output = `ls`;
```

In practice, the command menu and web shell redirect STDERR to STDOUT and capture the combined output from both streams by appending "2>&1" to the user-supplied command, as in the following example:

```
$output = `ls 2>&1`;
```

The exit code from the child process is automatically stored in the $? variable. This allows the CGI script to determine whether the remote application has executed successfully, and to display an error message to the user if not. However, if the child process fails to terminate, so will the CGI script, and the user will receive no feedback.

A second limitation of the web shell is its lack of support for environment variable management. As previously discussed, the CGI script is inherently stateless, and any environment variable set manually by the user via the web shell will be forgotten with the next invocation of the script. Memory of the current working directory across script invocations has been implemented as a special case; memory of the entire process environment is left for potential future work.

D. PROBLEMS ENCOUNTERED

This section describes problems that were not specific to the implementation of any single remote application support module, but rather that emerged as unforeseen properties of the remote application support system as a whole.

1. TRSS Access to Single-Level Network Interfaces

MYSEA servers are intended to interface not only with an MLS LAN, but also with various external single-level networks; remote applications making use of the MSKT interface are intended to have access to any of the external networks that correspond to their current session level, as well as to the MLS LAN. When connecting a MYSEA server to a single-level LAN, the security level of the network must be entered in the configuration data for the corresponding network interface on the server. (When connecting the MYSEA server to the MLS LAN, the corresponding interface is configured to accommodate the highest level of data permitted to pass through the MLS LAN.) Access to each network interface is monitored by a daemon running at the configured session level of the interface.

In order for the TRSS Child process to access the MLS LAN interface, it must run at the level of the MLS LAN. However, in order to support RA access to single-level networks, the TRSS Child must also be able to access network interfaces configured at lower levels. Even with MAC/DAC exemption enabled, this capability is not supported by the STOP 6 operating system.

Consequently, each interface on the MYSEA server must be configured at the level of the MLS LAN, regardless of the actual level of the connecting network. The enforcement of session level restrictions in communication between hosts is thereby transferred from the domain of the STOP 6 operating system to that of the MYSEA software. Specifically, the Peer Level Database must be correctly configured with the security levels of connected peers, and dutifully consulted by the TRSS Child before outgoing connections to single-level peers are permitted. This issue was not discussed in [3].

2. Trusted Parent Exemption

According to the specifications in [3], the TPS Child process is to launch the TRSS Parent process by means of the *xts_load_process* function provided by the STOP 6 operating system. However, it was discovered during the course of implementation that trusted programs with privileges cannot be loaded in this manner unless the loading process has operator-level integrity. Because the TRSS is a trusted program with privileges, and the TPS operates at the integrity level of the MLS LAN rather than operator integrity, the TPS failed to create the TRSS using the *xts_load_process* call.. Three potential solutions were considered:

1. Elevate the integrity level of the TPS process to operator integrity. This was an undesirable option because it violated the principle of least privilege; the TPS had no need for such a high level of integrity.
2. Implement the TRSS logic as a child process forked by the TPS Child, rather than as an external process loaded via *xts_load_process*. This was undesirable for two reasons:
    a. The resulting program would need to be assigned the union of the privilege sets required by the TPS and the TRSS, also in violation of the principle of least privilege.
    b. The TPS Child and TRSS Parent access different databases and have very different duties. The security engineering principles of data hiding and modularity therefore dictate that they be implemented separately.
3. Grant the TRSS the *Trusted Parent Exempt* privilege. This privilege allows it to be loaded by processes with less than operator-level integrity. The disadvantage of this approach is that the TRSS process may now be launched by any untrusted program.

After weighing each of the three options, it was decided that the third option was the most favorable. The risk of unauthorized execution of the TRSS Parent process was mitigated by setting the DAC controls on the TRSS executable such that it was accessible

only by the admin user. (The existence of rogue admin processes that might attempt to launch unauthorized TRSS processes was deemed acceptably improbable.) Because the TPS Child process runs as admin, it is able to launch the TRSS process as desired, while non-admin processes are denied access.

3. Multiple Binds

As discussed in Chapter III, any external connection established by a remote application must be registered in the Remote Connection Database in order for the SSS Parent to recognize it and handle incoming traffic for the connection appropriately. In order to register a connection in the database, it must have a known source port, source IP address, destination port, and destination IP address. This is problematic for certain socket calls, such as *connect* and *sendto*, that may be called on a socket that has not yet been bound to a source port and IP address.

To account for this, the TRSS Child process determines the appropriate source address to be used for a given destination address by consulting the Source Address Binding Database. It then makes the *bind* call on behalf of the remote application each time it requests an *mskt_connect* or *mskt_sendto* call.

This solution is imperfect, because when a *bind* call is requested for a socket that has been previously bound, subsequent *bind* calls for that socket fail. The specifications in [3] call for the *mskt_connect* and *mskt_sendto* functions to short-circuit their processing and return an error if their internal *bind* call fails, even if the failure is due to the fact that the socket was previously bound. This prevents remote applications making use of more than one instance of any of the *mskt_bind*, *mskt_connect*, or *mskt_sendto* functions from successfully completing their requested socket operations.

As a temporary solution, *bind* call failures with an errno of EINVAL (the error code returned when a socket has been previously bound) are ignored within the *mskt_connect* and *mskt_sendto* functions. This allows the functions to continue their processing and the RA socket operations to succeed.

For future work, a record should be maintained as to whether each socket entered in the MSKT Database has yet been bound. If a socket has already been bound, then subsequent *bind* calls on that socket should not be attempted.

E. SUMMARY

With the implementation of the components discussed in this chapter, remote application support in the multilevel MYSEA environment was enabled. Chapter V describes the developmental tests performed on each remote application support component implemented for this project, as well as the acceptance tests performed to ensure that the integrated remote application support system fulfilled its top-level user requirements. Chapter VI concludes with potential future work.

V. TESTING

A. OVERVIEW

This chapter describes the developmental and acceptance test plans designed to verify the proper functioning of the newly implemented RA support components. Three systems are utilized during testing: a remote-application-enabled MYSEA server, a Red Hat Linux system hosting a TFTP server, and a Windows XP client equipped with a Java TPE and web browser. Figure 8 depicts the setup of the test network. The MYSEA server has two network interfaces: one configured to connect to the MLS LAN on which the Windows client is located, and the other configured to connect to a single-level LAN housing the TFTP server. The single-level network is meant to simulate an external LAN that might operate at any one of various session levels; this level must be specified in the Peer Level Database on the MYSEA server. The network interfaces themselves are both configured at the level of the MLS LAN (at the maximum security level and integrity level 3, denoted *max:il3*), for the reasons discussed in the previous chapter.

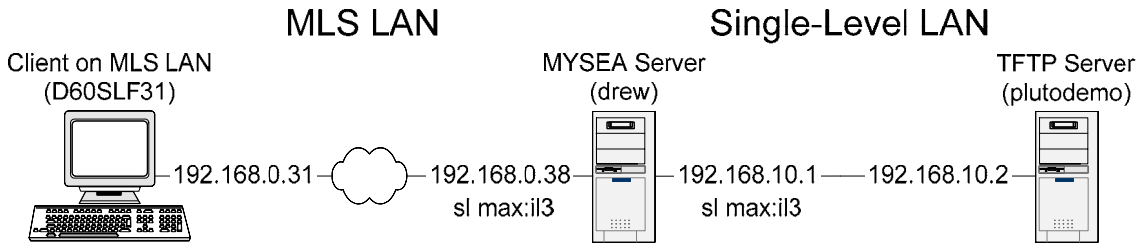

Figure 8. Test Network Topology

The developmental and acceptance test plans and results are discussed at a general level in this chapter; specific test procedures are documented in Appendix C.

B. DEVELOPMENTAL TESTING

The purpose of developmental testing is to test the functionality of each of the components of remote application support implemented or modified as part of this thesis.

These include the TRSS Parent and Child processes that execute socket calls on behalf of remote applications, the CGI scripts that invoke the remote applications, and the remote applications themselves.

Reproductions of the Process Overview figure (Figure 1) are presented again within each of the following sections as a reminder of the context in which each RA component of interest was designed to function.

1. Trusted Remote Session Server (TRSS)

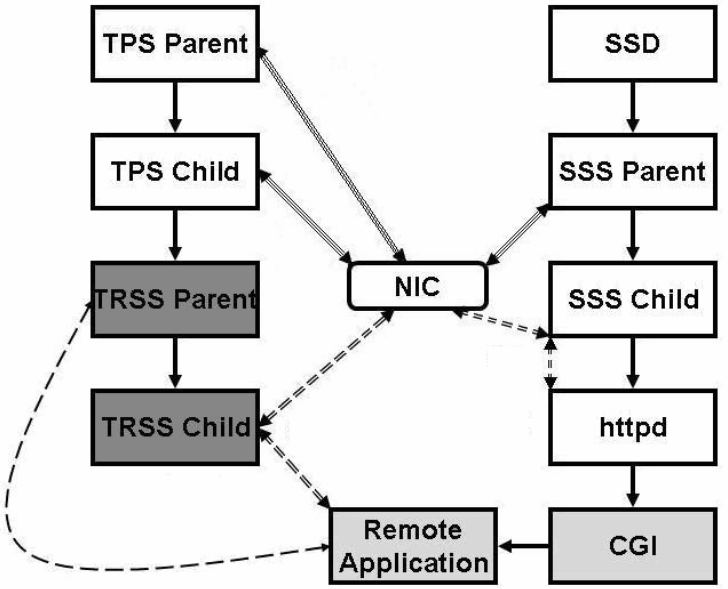

Figure 9. TRSS Processes in the Context of Remote Application Support

The purpose of the first test suite is to verify that the TRSS Parent and Child processes correctly handle the complete set of MSKT socket calls that may be requested by a remote application. Although the functionality of the TRSS processes is tested implicitly in later tests of the remote application support system as a whole, this test suite is the only one to provide complete coverage of supported MSKT socket calls.

For these tests, an MSKT-enabled test server is remotely invoked on the MYSEA server via the web shell, and a local client is launched on the MYSEA server to interact with the test server. Although both client and server run on the MYSEA server, only the

server runs as a remote application and makes use of MSKTs.[3] The client presents a menu of user-selectable options, each of which causes specific MSKT socket operations to be exercised by the server. The *mskt_socket*, *mskt_bind*, and *mskt_close* calls are exercised implicitly. Together, the tests exercise every one of the supported MSKT socket calls.

Note that this test suite was designed to test only the logic implemented specifically within the TRSS Parent and Child source code, and not to test the previously implemented RA support databases and modules that the TRSS processes depend on. The formal test documentation for these components is provided in [17].

The client and server used for testing were implemented prior to the start of this thesis, and were used for this test suite almost entirely without modification. The single change required was in the "Miscellaneous testing" function, which was designed to test the *mskt_getpeername*, *mskt_getsockname*, *mskt_getsockopt*, *mskt_setsockopt*, *mskt_fcntl*, and *mskt_ioctl* calls. The *mskt_ioctl* function takes as its first and second parameters a file descriptor and an integer representing an action to be taken on that file descriptor. To ensure that the function could be handled without error on the part of the TRSS Child, a valid *ioctl* action request for XTS-400 network sockets had to be supplied. (The *ioctl* request made by the original test server returned a value of -1 when applied to XTS-400 network sockets. This was not useful, because an *mskt_ioctl* return value of -1 does not distinguish a failure of the *ioctl* call from a failure of the TRSS Child process.) With some experimentation, it was determined that the *ioctl* FIONREAD request could be successfully applied to network sockets on the XTS-400, and the test server was modified to make this request.

For the *mskt_ioctl* call to be properly tested in the future, the test server should be modified so that the actual functionality of the FIONREAD request (and all other supported types of *mskt_ioctl* socket requests) may be verified. All that matters for the purposes of TRSS testing is that the *mskt_ioctl* call returns without an error, indicating

[3] The current design for remote application support allows for the possibility of either client or server RAs running on the MYSEA server, although the client RA would seem the more natural case. The server RA used in this test suite exercises all of the supported client-side socket calls as well as all of the server-side socket calls, and is therefore ideally suited to the task of testing the TRSS processes' handling of the full range of MSKT socket calls.

that the TRSS Child has successfully received the request from the test server, made the *ioctl* call on its behalf, and returned the result. Whether the *ioctl* call itself functions as intended is beyond the scope of TRSS testing.

In the following test suite table, the "Action" column indicates the menu option selected within the client application for each test case, and the "RA" column specifies the process running as the remote application (the test_socket_ra server in this test suite). Both the client and the server generate reports as the tests are run; output constituting a success for each test case is documented in Appendix C.

Figure 9 highlights the TRSS processes within the context of remote application support.

Test ID	Test Type	Action	RA	Expected Result
a1	read	Test *mskt_read*	test_socket_ra	Success
a2	write	Test *mskt_write*	test_socket_ra	Success
a3	select	Test *mskt_select*	test_socket_ra	Success
a4	listen	Test *mskt_listen*	test_socket_ra	Success
a5	accept	Test *mskt_accept*, *mskt_connect*	test_socket_ra	Success
a6	shutdown	Test *mskt_shutdown*	test_socket_ra	Success
a7	send	Test *mskt_send*	test_socket_ra	Success
a8	sendto	Test *mskt_sendto*	test_socket_ra	Success
a9	recv	Test *mskt_recv*	test_socket_ra	Success
a10	recvfrom	Test *mskt_recvfrom*	test_socket_ra	Success
a11	fork	Test *mskt_fork*	test_socket_ra	Success
a12	Blocked I/O	Test *mskt_fcntl*	test_socket_ra	Success

Test ID	Test Type	Action	RA	Expected Result
a13	Misc testing	Test *mskt_getpeername, mskt_getsockname, mskt_getsockopt, mskt_setsockopt, mskt_fcntl, mskt_ioctl*	test_socket_ra	Success

Table 5. TRSS Testing

2. CGI Invocation of Remote Applications

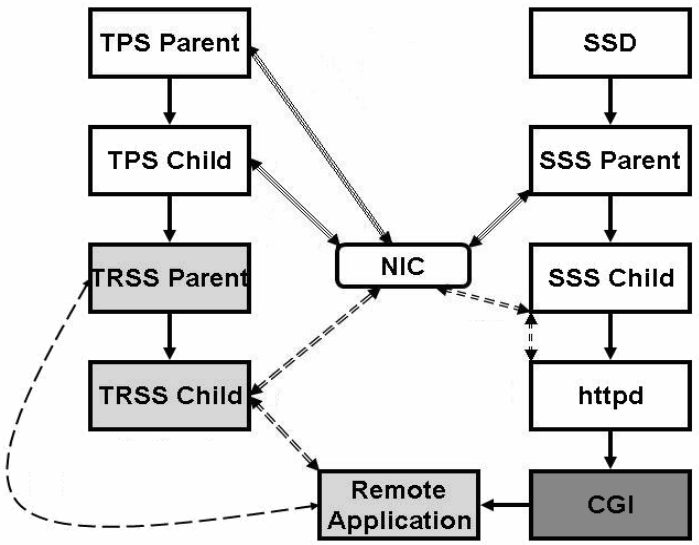

Figure 10. CGI RA Invocation Processes in the Context of Remote Application Support

The purpose of this test suite is to verify the functionality of the CGI mechanism for executing remote applications. Each of the two CGI interfaces is tested: the web shell, which allows users to enter arbitrary UNIX-style commands to be executed on the server, and the Windows-style command menu, which allows users to select desired actions from a list. In each case, the CGI interface is invoked via a web browser on the MLS client, and the CGI script and requested command are executed on the MYSEA server. The output of the command is returned via a web page back to the client.

The CGI RA Invocation processes are highlighted within the context of remote application support in Figure 10.

a. Web Shell Functional Testing

Functional testing of the web shell is meant to verify that the shell may be used to successfully invoke user-specified remote applications on the MYSEA server and report the results back to the user. Testing should also verify that the remote application is executed at the session level and with the user ID of the remotely authenticated user. Because the change-directory (`cd`) command is implemented as a special function of the CGI script and not executed directly as a remote application, it is tested explicitly.

These tests are performed by logging in to the MYSEA server from the MLS client, running a session, navigating to the URL of the web shell, and interacting with the server via a text input field on the web page.

Test ID	Test Type	Action	RA	Expected Result
b1	Web shell directory-listing check	List contents of a directory via the web shell	ls	Contents of directory are displayed
b2	Web shell change-directory check	Change directories via the web shell	(none)	New working directory is confirmed
	Web shell directory-listing check after change-directory	List contents of current directory via the web shell	ls	Contents of new working directory are displayed
b3	Web shell user ID check	Check user ID via the web shell	whoami	User ID entered by the user at login is displayed
b4	Web shell session-level check	Check session level via the web shell	mysea_level	Session level previously negotiated by the user via the TPE is displayed

Table 6. Web Shell Functional Testing

b. Web Shell Exception Testing

The purpose of exception testing is to verify that the web shell does not demonstrate unintended behavior when presented with unexpected or ill-formed user input. This may include an accidental request to invoke a remote application that does not exist, or an attempt to "break" the script by manually forging form input names or values. Testing should verify that the web shell is functional even after being provided with invalid input.

To demonstrate more concretely the purpose of this test suite, an excerpt from the HTML source composing the web shell is presented below:

```
<form …>
…
Command: <input type=text size=40 name=cmd value=''>
…
</form>
```

This excerpt encodes an HTML form with a single input name/value pair, namely, an input entitled "cmd," which takes the form of a text box and has no pre-set value. When a user types a command into the text box and submits the form, the string he or she enters in this text box becomes the value of the "cmd" input. When the CGI script on the server is invoked, it parses the user input into name/value pairs, searches for the input entitled "cmd," and interprets the corresponding value to be the name of the remote application the user wishes to execute.

There are three main possibilities for error in this setup. First, the user may enter an invalid remote application name as the value of the "cmd" input. Test c1 verifies that the CGI script prints an informative error message and returns a new, functional shell in this case.

The second possibility is that the user may enter no value for the "cmd" input. Test c2 verifies that the CGI script re-displays the original web shell in this case.

The final possibility is that the user may submit a form without a “cmd” input at all, and possibly with some other input type instead. Note that this is different from simply leaving the “cmd” text field empty and submitting the form; in that case, the script still receives the “cmd” name/value pair, but the value is null. In the present case, the script does not receive a “cmd” name/value pair at all.

The choice to implement the CGI scripts using the HTTP POST method, rather than HTTP GET, was made in an attempt to minimize the possibility of this third type of error occurring by accident. Forms implemented using the HTTP GET method reveal their input name/value pairs in their URLs. For example, after entering “ls” in the “cmd” field of the web shell, the new URL displayed in the address bar would take the form *http://servername/cgi-bin/webshell.cgi?cmd=ls*. It would be easy for an adventurous but non-malicious user to alter the input name/value pairs by manually editing the URL, and to do so in a way that was unexpected by the CGI script. Using the HTTP POST method hides the input names and values from the user, so that in order to submit an invalid input name or omit an expected input name, the user would have to manually edit the web shell source HTML or write a special-purpose client to interact with the CGI script, neither of which is likely to be accidental.

Test c3 verifies that the submission of an invalid form input name/value pair in place of the expected “cmd” input does not result in unintended behavior by the web shell. By design, the script searches only for the value of the “cmd” input and re-prints the original shell if no such value is be found. Since we have established that this type of error is not likely to happen by accident, informative error reporting is not as vital in this case as in others.

Test ID	Test Type	Action	RA	Expected Result
c1	Invalid web shell command (i.e., invalid value for "cmd" input field)	Attempt to execute a non-existent RA via the web shell	(none)	Error message and new command prompt are displayed
	Web shell functionality check after invalid input	List contents of a directory via the web shell	ls	Contents of directory are displayed, indicating that web shell is still functional
c2	Empty web shell command (i.e., null value for "cmd" input field)	Leave "cmd" field empty and submit form	(none)	New command prompt is returned
	Web shell functionality check after invalid input	List contents of a directory via the web shell	ls	Contents of directory are displayed, indicating that web shell is still functional
c3	Invalid form input name instead of expected "cmd"	Submit request with manually forged form input name instead of expected "cmd"	(none)	New command prompt is returned with no output from invalid request
	Web shell functionality check after invalid input	List contents of a directory via the web shell	ls	Contents of directory are displayed, indicating that web shell is still functional

Table 7. Web Shell Exception Testing

c. Command Menu Functional Testing

The purpose of the command menu functional testing is to verify that the command menu interface may be used to successfully invoke remote applications

selected by the user from a list and report the results back to the user. Each of the options presented in the menu is tested. Because the command menu CGI script is invoked using exactly the same mechanisms and trusted processes as the web shell, explicit verification of its user ID and session level need not be performed here.

Test ID	Test Type	Action	RA	Expected Result
d1	Command menu directory-listing check	List contents of a directory via the command menu	ls	Contents of directory are displayed
d2	Command menu file-contents-display check	Display contents of a text file via the command menu	cat	Contents of file are displayed
d3	Command menu change-directory check	Change directories via the command menu	(none)	New working directory is confirmed
	Command menu directory-listing check after change-directory	List contents of current directory via the command menu	ls	Contents of new working directory are displayed
d4	Command menu move check	Move a file via the command menu	mv	Move request is confirmed with no errors reported
	Command menu directory-listing check after move	List contents of directory containing moved file via the command menu	ls	Moved file is displayed in its new location and not its old location
d5	Command menu copy check	Copy a file via the command menu	cp	Copy request is confirmed with no errors reported

Test ID	Test Type	Action	RA	Expected Result
	Command menu directory-listing check after copy	List contents of directory containing the copied file via the command menu	ls	Copied file is displayed in its new location and its old location
d6	Command menu web check	Request a web page via the command menu	swget	Requested web page is displayed
d7	Command menu TFTP check	Issue TFTP GET request via the command menu	tftp	TFTP reports that file was transferred
	Command menu file-contents-display check after TFTP	Display contents of requested file via the command menu	cat	Contents of requested file are displayed

Table 8. Command Menu Functional Testing

d. Command Menu Exception Testing

The purpose of exception testing is to verify that the command menu does not demonstrate unintended behavior when presented with unexpected or ill-formed user input. This may include attempts to execute unlisted remote applications or to "break" the script by manually forging form input names or values as described earlier in this chapter, or by supplying specially crafted, non-alphanumeric command-line arguments as form inputs. To prevent users from executing unsupported remote applications (e.g., by appending a semi-colon to a command argument and following it with a new command, which a normal shell would interpret as two separate commands), the CGI script verifies that command arguments contain only the following types of characters: letters, numbers, slashes, periods, underscores, and dashes.

The following tests verify that the command menu is functional even after being provided with invalid input.

Test ID	Test Type	Action	RA	Expected Result
e1	Invalid command (i.e., invalid form input value for “cmd” input field)	Attempt to execute an unsupported RA via the command menu	(none)	“Illegal input” error message is displayed
	Command menu functionality check after invalid input	List contents of a directory via the command menu	ls	Contents of directory are displayed, indicating that command menu is still functional
e2	Empty command (i.e., null form input value for “cmd” input field)	Select no option and submit form	(none)	New command menu is returned
	Command menu functionality check after invalid input	List contents of a directory via the command menu	ls	Contents of directory are displayed, indicating that command menu is still functional
e3	Invalid form input instead of expected “cmd”	Submit request with manually forged form input name instead of expected “cmd”	(none)	New command menu is returned with no output from invalid request
	Command menu functionality check after invalid input	List contents of a directory via the command menu	ls	Contents of directory are displayed, indicating that command menu is still functional
e4	Non-alphanumeric arguments	Submit arguments containing forbidden punctuation	(none)	Error message is displayed
	Command menu functionality check after invalid input	List contents of a directory via the command menu	ls	Contents of directory are displayed, indicating that command menu is still functional

Test ID	Test Type	Action	RA	Expected Result
e5	Incorrect number of arguments	Attempt to move a file without specifying source and destination locations	mv	Error-handling is delegated to RA (RA reports an error)
	Command menu functionality check after invalid input	List contents of a directory via the command menu	ls	Contents of directory are displayed, indicating that command menu is still functional

Table 9. Command Menu Exception Testing

3. Remote Applications

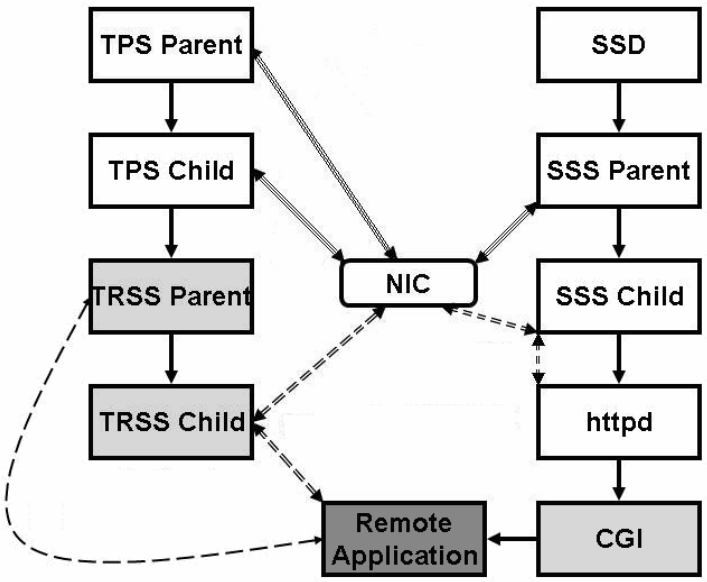

Figure 11. TFTP Client and Swget in the Context of Remote Application Support

The purpose of this test suite is to verify the functionality of the proof-of-concept remote applications installed on the MYSEA server. These include a publicly available TFTP client altered to use MSKT socket calls, and a simple web client (swget) that uses MSKT socket calls to retrieve requested web pages. In this test suite, the remote applications are invoked using the web shell.

Once again, this test suite was designed to test only the logic implemented or added specifically to the remote application source code. It was not meant to exhaustively test the implementation of the databases and modules that the remote applications depend on.

The RA processes are highlighted within the context of remote application support in Figure 11.

a. TFTP Client Functional Testing

The purpose of the TFTP client functional testing is to verify that the MSKT-enabled TFTP client can be used to successfully transfer files from a TFTP server using the TFTP GET request. It should also verify that the file transfer and termination of the TFTP client can be accomplished via a single command-line instruction, functionality that was added for the purpose of easily integrating the TFTP client with the CGI remote application invocation mechanism. In this test, the user requests a file transfer via the web shell, and verifies that the requested file was successfully transferred into the specified directory on the MYSEA server. The file is then displayed to ensure that it contains the expected content.

Test ID	Test Type	Action	RA	Expected Result
f1	Non-interactive TFTP file request	Issue TFTP GET request via web shell	tftp	TFTP client reports that file was transferred
		Display file (specifying the local path included in the original TFTP request)	cat	Contents of requested file are displayed, and match known content of file

Table 10. TFTP Client Functional Testing

Because the TFTP client is based on publicly available software, exception testing is omitted here.

b. Web Client Functional Testing

The purpose of the web client (swget) functional testing is to verify that the MSKT-enabled web client can be used to successfully transfer files from an HTTP server. A valid request should succeed whether or not the requested resource is prefixed with "http://".

Test ID	Test Type	Action	RA	Expected Result
g1	Valid web page request with HTTP prefix	Request valid URL with "http://" prefix	swget	Markup content of requested web page is displayed
g2	Valid web page request without HTTP prefix	Request valid URL without "http://" prefix	swget	Markup content of requested web page is displayed

Table 11. Web Client Functional Testing

c. Web Client Exception Testing

Exception testing of swget is designed to verify that the program provides informative error messages when a requested file cannot be retrieved or when provided with an incorrect number of command-line arguments.

Test ID	Test Type	Action	RA	Expected Result
h1	Too few command-line arguments	Invoke swget with no command-line arguments.	swget	Usage error message is displayed
h2	Too many command-line arguments	Invoke swget with two command-line arguments.	swget	Usage error message is displayed
h3	Unavailable server	Request a file from an unavailable server.	swget	Connection error message is displayed
h4	Unavailable file	Request an unavailable file from an available server.	swget	Server error message is displayed

Table 12. Web Client Exception Testing

C. DEVELOPMENTAL TESTING RESULTS

Developmental testing of the CGI scripts revealed no unforeseen problems. The scripts did not need to be tailored for the XTS-400 or MYSEA software, and were straightforward to implement and test.

Developmental testing for the TRSS processes and remote applications was slightly more revealing, with a handful of factors contributing to initial failures:

- The problems described in Chapter IV regarding TRSS access to single-level networks, socket call failures due to multiple binds, and *xts_load_process* failures were unanticipated and required investigation.

- Minor bugs in previously implemented, but untested MYSEA remote application support modules had to be pinpointed and corrected.

- Because the cleanup mechanisms for remote application support were still in the design phase and not fully implemented, the presence of stale processes and data structures meant that the server daemons often had to be restarted, or the

MYSEA server rebooted entirely, between each test. (Cleanup issues are discussed in more detail in the "Future Work" section of Chapter VI.) Furthermore, each time a change was made to the source code of a trusted program, it not only had to be recompiled, but the trusted binary replaced and (if applicable) the daemon restarted. This made it a very time-consuming process to correct problems and re-run tests.

Once these initial stumbling blocks were surmounted, developmental testing successfully demonstrated the functionality of each of the components implemented as part of this project. The results of all test suites, including those testing the TRSS Parent and Child processes, the CGI remote application invocation mechanisms, and the MSKT-enabled TFTP and swget clients, were as expected. Detailed test results are provided in Appendix C.

D. ACCEPTANCE TESTING

The goal of acceptance testing is to verify that the newly implemented remote application components are able to successfully interact with each other in such a way that the top-level user requirements for remote application support are fulfilled. These requirements, first stated in Chapter III B, are as follows:

1. The RA shall be able to communicate with the local MLS server, a remote MLS server and a RA server.
2. The remote application shall be appropriately bound to the authenticated user's session. Specifically, the RA process shall run with the user ID and at the current session level of the authenticated user.
3. The user shall be able to launch the RA from the client.
4. The MLS server shall be able to support both Unix/Linux and Microsoft Windows clients.
5. The design shall only require a minimal number of changes to the RA.

Of these, requirements 2 and 3 are demonstrated to be fulfilled in developmental testing, and requirements 4 and 5 are fulfilled by design. (Although our testing is conducted from a Windows client, there is no reason to believe that results would differ for a Unix/Linux client, since remote application invocation is accomplished via a platform-neutral web page.) This leaves only the first requirement, which specifies three test cases:

1. Communication between a remote application and an external single-level server.
2. Communication between a remote application and the local MLS server.
3. Communication between a remote application and a remote MLS server.

Of these, the first and second test cases are described in this section; the third test case was determined to be unrealizable in the current MYSEA environment, and is described in the "Future Work" section of Chapter VI.

The purpose of the functional acceptance testing is to verify that in Test Cases 1 and 2, the remote applications are able to successfully retrieve the requested data, with the stipulation that the user making the request must be logged in at an allowable session level relative to the peer hosting the files (or the files themselves, if hosted on a multilevel server).

The purpose of the exception testing is to verify that file transfer requests that are not allowable under the Bell-LaPadula security model or the Biba integrity model are in fact denied.

1. Communication between an RA and an External Server

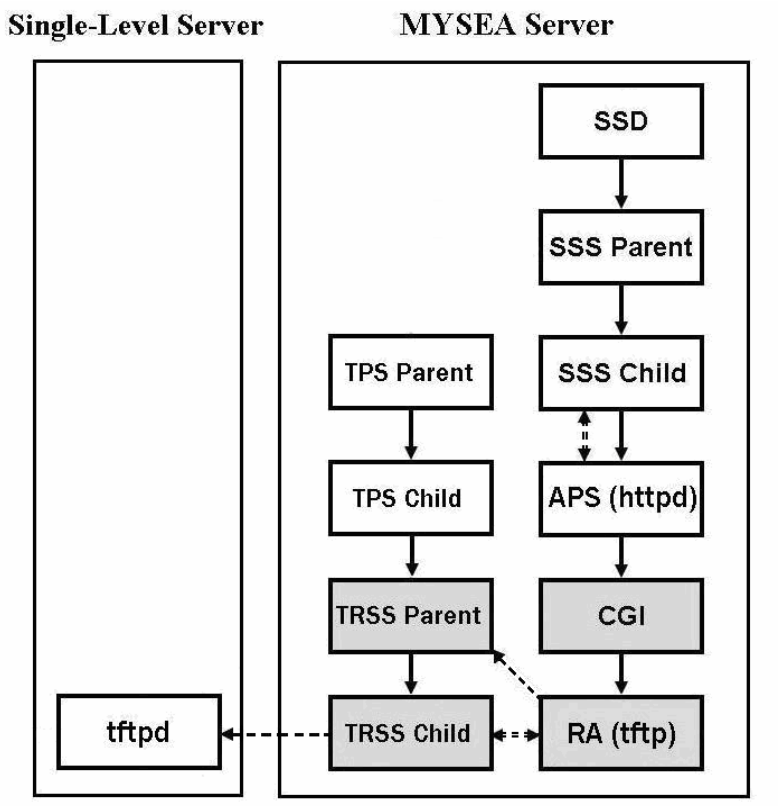

Figure 12. RA Request to External Server

Acceptance Test Case 1 involves communication between a remote application and an external, single-level server. It verifies that a remotely invoked TFTP client running on the MYSEA server is able to successfully request data from a single-level peer if the level of the peer is equal to the session level of the requesting user, but not otherwise. This tests the proper configuration of the Peer Level Database and its use by the TRSS Child process in determining whether to allow the requested *sendto* and *recvfrom* socket calls.

Because a successful file request and transfer involves two-way information flow between the client and server, both the simple security policy and the security-* property of the Bell-LaPadula model apply in determining whether a file request will succeed. The simple security policy limits allowable object (peer) security levels to those less than or equal to the security level of the subject, while the security-* property limits allowable

peer levels to those greater than or equal to the security level of the subject. The STOP 6 implementation of the security-* property further limits allowable objects levels to those exactly equal to the security level of the subject. Taken together, these properties restrict the security level of the requesting user to precisely the security level of the peer hosting the requested file. The simple integrity policy and integrity-* property of the Biba integrity model similarly restrict the integrity level of the requesting user to precisely the integrity level of the peer hosting the requested file. However, integrity-level checking has not yet been implemented in the Utility Module access functions relied upon by the TRSS Child, and is therefore not tested here.

In this test suite, a user logged in to the MYSEA server from a client on the MLS LAN remotely invokes the TFTP client and requests the transfer of test files from an external TFTP server to the MYSEA server. The RA-related processes involved in this interaction are depicted in Figure 12. The test network topology from Figure 8 is presented again below for reference.

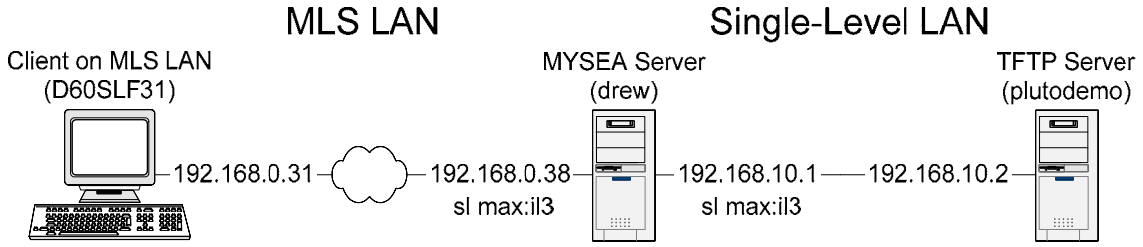

Figure 13. Network Components Involved in Acceptance Test Case 1

a. Acceptance Test Case 1 Functional Testing

Functional testing for this test case verifies that the TFTP file request is successful when the security level of the user's current session is equal to the security level of the TFTP server. For this test, the TFTP server is registered at the sl1:il3 level in the Peer Level Database, and the user logs in to the MYSEA server at the same level. The user then requests the transfer of a sample text file from the TFTP server to the MYSEA server and displays the contents of the transferred file.

Tests are conducted via the CGI command menu.

Test ID	Test Type	User Session Level	Peer Level	Action	RA	Expected Result
i1	Secrecy read-equal/write-equal	sl1:il3	sl1:il3	Issue TFTP GET request via command menu	tftp	TFTP client reports that file was transferred
				Display file via command menu	cat	Contents of requested file are displayed

Table 13. Acceptance Test Case 1 Functional Testing

b. Acceptance Test Case 1 Exception Testing

Exception testing for this case verifies that the TFTP file request fails if the security level of the user's current session is not equal to the security level of the TFTP server, or if the security level of the TFTP server is unknown. For each of the tests in this suite, the TFTP server is registered in the Peer Level Database at a security level different from the user's current session level (or is not registered in the database at all). In these cases, the user's file transfer request should result in an error message and the file should not be transferred.

Test ID	Test Type	User Session Level	Peer Level	Action	RA	Expected Result
j1	Secrecy read-up	sl1:il3	sl2:il3	Issue TFTP GET request via command menu	tftp	"Permission Denied" error is displayed
				Display file via command menu	cat	File contents are empty[4]
j2	Secrecy write-down	sl1:il3	sl0:il3	Issue TFTP GET request via command menu	tftp	"Permission Denied" error is displayed
				Display file via command menu	cat	File contents are empty
j3	Peer-level-unknown	sl1:il3	Undefined	Issue TFTP GET request via command menu	tftp	"Permission Denied" error is displayed
				Display file via command menu	cat	File contents are empty

Table 14. Acceptance Test Case 1 Exception Testing

[4] The TFTP client opens the local target file for writing before it even attempts to make a socket connection, so an empty file is created even if the connection is disallowed.

2. Communication between an RA and a Local APS

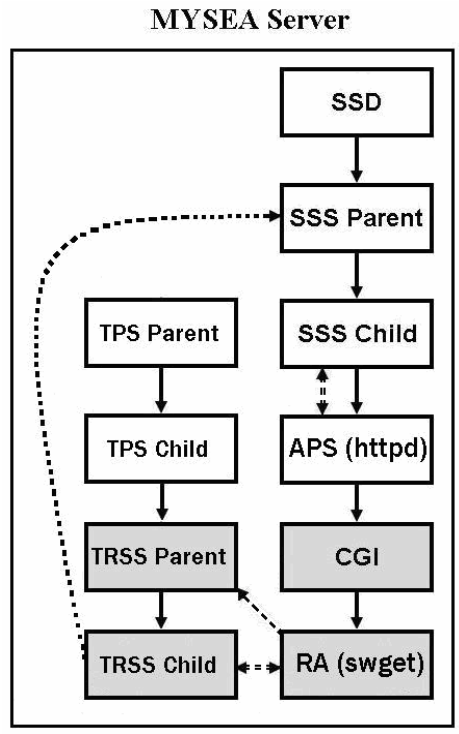

Figure 14. RA Request to Local APS via SSS

Acceptance Test Case 2 verifies that a remotely invoked web client running on the MYSEA server is able to successfully request data from a web server (*httpd*) running as an APS on the same system. This tests the proper configuration of the Peer Level Database and its use by the TRSS Child Process in determining whether to allow the requested socket *connect* call. In this case, the Peer Level Database should indicate that the peer is multilevel, and the TRSS Child Process should therefore allow the connection.

This test case also verifies the proper configuration and use of the Remote Connection Database by the SSS Parent and TRSS Child processes. Before the TRSS Child may establish connections on behalf of remote applications, it is required to register those connections in the Remote Connection Database. The SSS Parent refers to the database when attempting to validate an incoming request (such as a web page request) that does not originate from the TPE of an authenticated user. If the connection is found in the Remote Connection database, the SSS Parent retrieves the session level and user

ID associated with the connection in the database and the SSS Child creates an APS process to service the request. The APS process is created at the same session level and with the same user ID as the registered remote connection, and is therefore subject to the same access control restrictions as the requesting user as it attempts to access web pages of various security and integrity levels. These tests will verify that remote connections and their corresponding session information are being successfully entered and retrieved from the Remote Connection Database. The RA-related processes involved in this interaction are depicted in Figure 14.

In this test suite, a user logged in to the MYSEA server from a client on the MLS LAN remotely invokes a web client and issues web page requests to the local web server. Tests are conducted via both the command menu and the web shell. When requesting a web page via the command menu, an HTML-rendered version of the requested page is displayed in lieu of the original menu; when making the request via the web shell, the raw markup of the requested file is displayed within the emulated terminal window.

The network components involved in this test case are depicted in Figure 15.

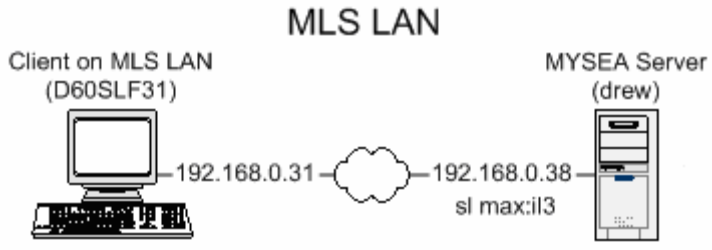

Figure 15. Network Components Involved in Acceptance Test Case 2

a. Acceptance Test Case 2 Functional Testing

Functional testing for this test case should verify that a remote user may launch a web client on the MYSEA server which is able to successfully request a local web page via the SSS. The test also verifies that the instance of the web APS invoked to handle the requests runs at the session level and with the user ID of the requesting user. Because the APS is untrusted, its session level and user ID limit the files it is able to read to those allowable under the MAC and DAC policies of the STOP 6 operating system;

this has been demonstrated outside of the scope of remote application support and is therefore not formally demonstrated here.

In this test suite, the user requests a dynamically generated web page that displays the current session level and user ID of the CGI (and therefore, the APS) process that generated it, along with the server IP address, the IP address of the requesting host, and the time of the request. This information is used to verify that the web request received by the APS was generated by the MYSEA server and not by the remote client, and also that the APS is running with the user ID and at the session level of the remote user.

Test ID	Test Type	User Session Level	Action	RA	Expected Result
k1	Web page request (sl1:il3)	sl1:il3	Request CGI-generated web page via command menu	swget	Rendered version of requested web page is displayed; requesting IP address is that of MYSEA server; session level of process is sl1:il3
k2	Web page request (sl3:il3)	sl3:il3	Request CGI-generated web page via command menu	swget	Rendered version of requested web page is displayed; requesting IP address is that of MYSEA server; session level of process is sl3:il3

Table 15. Acceptance Test Case 2 Functional Testing

b. Acceptance Test Case 2 Exception Testing

Exception testing for this case verifies that the web page request fails if the MYSEA server was not registered in its own Peer Level Database as being an MLS peer. For this test, the MYSEA server administrator removes the record from the database, and a user requests a web page from the MYSEA server via the command menu. The request should result in a connection error message, and the web page should not be displayed.

Test ID	Test Type	User Session Level	Peer Level	Action	RA	Expected Result
l1	Peer-level-unknown test	sl1:il3	Undefined	Request web page via command menu	swget	Connect error message is displayed

Table 16. Acceptance Test Case 2 Exception Testing

E. ACCEPTANCE TESTING RESULTS

The results of the acceptance tests were as expected, successfully demonstrating that remote applications are able to communicate with local MLS servers and external single-level servers when such communications do not violate the access control policies of the multilevel environment. With the exception of the third envisioned usage scenario (communication between RAs and remote MLS servers, left for future work), acceptance testing has demonstrated that the current implementation of the remote application support mechanism meets the top-level user requirements documented in [3]. Detailed test results are provided in Appendix C.

F. SUMMARY

This chapter has described the developmental tests performed on each remote application support component implemented for this project, and the acceptance tests performed to ensure that the integrated remote application support system fulfills its top-level user requirements. All tests were successful.

Chapter VI concludes with a project summary and suggestions for future work.

VI. CONCLUSION

A. SUMMARY

This project has enhanced the usability of the MYSEA architecture by providing users the ability to execute server-resident applications from diskless MYSEA clients. Specifically, the project has involved the implementation of the following remote application support components:

1. The Trusted Remote Session Server (TRSS) processes responsible for making socket calls on behalf of untrusted remote applications;
2. Two distinct Common Gateway Interface (CGI) interfaces for invoking remote applications from a client web browser:
    a. A command menu, from which a user may select desired remote applications to be executed on the server;
    b. A web shell, through which a user may enter UNIX-style commands to be executed on the server.
3. Two simple network-enabled remote applications:
    a. A TFTP client, adapted from publicly available source code [14];
    b. A web client, developed specifically for use as a remote application on the MYSEA server.

This project has also involved the development and execution of a test plan to verify the functionality of the newly implemented remote application support components. Testing has successfully demonstrated the functionality of each component and of the remote application support system as a whole.

B. ANALYSIS OF THESIS QUESTIONS

Two research questions were posed at the beginning of this project:

1. What modifications to the existing design are necessary to successfully implement remote application support on the MYSEA server?

2. What additional functionality, if any, is required to support the remote execution of specific desired applications?

The answer to the first question is that surprisingly few modifications were required. The specifications developed in [3] were sufficiently detailed and thorough that a newcomer to the project was able to pick up where the designer left off and implement a functional remote application support system without significant difficulties. The refinements that were required fell into two main categories:

- *Privilege and access issues specific to the STOP 6 operating system.* These are issues that would have been difficult to foresee without actually attempting an implementation. Issues that fell under this category included:
    - *The inability of the TRSS process to access single-level network interfaces configured at levels lower than itself.* One might reasonably have assumed that enabling MAC/DAC exemption on the part of the TRSS process would solve the problem, but this was not the case.
    - *The inability of the TPS Child process to load the TRSS Parent process without assigning the TRSS process the Trusted Parent Exemption.* It would have been difficult to anticipate the restriction against the invocation of privileged processes by processes below operator-level integrity without specifically searching the documentation for such restrictions.
- *Issues involving connectionless protocols.* Remote applications that make use of connectionless protocols such as UDP do not necessary call *close* or *shutdown* on their sockets at the end of data transfer sessions. The author of [3] identified one potential consequence of this fact: connections that are entered in the Remote Connection Database when these applications make their *mskt_sendto* calls may remain in the database as "zombie" entries, since it is normally the *mskt_close* and *mskt_shutdown* calls that prompt the removal of the entries from the database. Two other issues involving connectionless protocols were identified during this project:

- *Socket call failures due to multiple binds.* *bind* is called internally for every *mskt_bind*, *mskt_connect*, and *mskt_sendto* call requested by a remote application; binding the same socket more than once will result in the failure of all bind calls after the first, and of the MSKT socket calls in which they are attempted. This prevents remote applications making use of more than one instance of any of the *mskt_bind*, *mskt_connect*, or *mskt_sendto* functions from successfully completing their requested socket operations. While this might not be a problem for TCP clients that make a single *connect* call followed by only *send* and *recv* calls, or for TCP servers that make a single *bind* call followed by *listen*, *accept*, and *send* calls, it is extremely problematic for UDP clients that typically make repeated *sendto* calls, since each one after the first will fail. A temporary fix was implemented for this project, but a more robust solution (such as the one suggested in Chapter IV) should be implemented for future work.

- *Zombie TRSS processes.* A TRSS Child process should not live longer than the remote application that it was invoked to serve. Under the current specifications, the TRSS Child process terminates when its remote application calls *mskt_close* or *mskt_shutdown* on its final open socket. However, if the remote application uses a connectionless protocol and never makes either of these calls, the TRSS Child will continue running on the server as a zombie process. The same potential issue exists for SSS Child processes, which should not live longer than the APS processes that they serve. (On the flip side, a remote application may close its final socket but still have a significant amount of processing left to do; it does not necessarily make sense for the TRSS Child to terminate it as soon as it has closed its final socket. This issue is being addressed separately)

This leads us to the second question: what additional functionality is required to support the remote execution of specific desired applications?

For the purposes of this project, none. A TFTP client and simple web client were each successfully adapted for use as remote applications using the precise instructions provided in [3]. There were, in the case of the UDP-based TFTP client, issues with zombie TRSS processes, but the clients were otherwise functional. Cleanup issues in general will need to be addressed in future work.

Because the applications ported for this project were relatively simple, it is possible that problems will emerge for more complex applications – those that make use of unsupported *ioctl* or *fcntl* calls, for example. There is also the question as to how best to handle remote applications that make library calls which in turn make socket calls, since these socket calls cannot be mapped to their MSKT equivalents by simply editing the source code of the remote applications. These are important issues for future work.

C. FUTURE WORK

Additional areas for future work include the following:

1. Federated Server Environment

The top-level user requirements for remote application support, documented in [3], specify that a remote application should be able to make APS requests to a remote MYSEA server. However, the SSS Parent process that validates incoming APS requests was designed to accept only two types of connections: those originating from users logged in to the server via a TPE, and those originating from a remote application executing on the server. When the SSS Parent receives a request, it checks for the requesting entity in the User and Remote Connection Databases, and if it does not find a record of it in either database, it drops the connection.

When a remote connection makes an outgoing connection request, it is entered into the Remote Connection Database on the server on which it executes; the remote application may therefore make APS requests of the local server, and the SSS Parent can verify that the requests originate from a valid remote application. However, if the remote

application attempts to make a connection with a remote MYSEA server, that server will have no record of the connection request in its own Remote Connection Database, and the SSS Parent will drop the request. It is therefore impossible in the current MYSEA environment for a remote application to communicate with an APS on a remote MYSEA server.

Visions for the future include a federated server environment, in which MYSEA servers will share databases containing information about authenticated users and remote connections. In this environment, connections registered in the Remote Connection Database by one MYSEA server will be readable by remote MYSEA servers. This will enable the SSS Parent process to validate requests originating from remote applications executing not just on its own server, but on any federated MYSEA server.

2. Stress Testing

The very nature of remote application support creates the potential for a performance bottleneck at the server hosting the remote applications. In the current implementation of remote application support on the MYSEA server, several of the databases shared between remote application support processes are locked by the process accessing them, in order to prevent race conditions. This could exacerbate performance bottleneck issues. Affected databases could include the RAMSKT Map, APSMSKT Map, MSKT, Remote Connection, and Cleanup Databases. Stress testing should be performed to check for performance degradation as the number of simultaneous users and APS and RA requests increases.

3. Cleanup

When a user logs out or changes session level, the data structures and processes that provided remote application support for the user's previous session should be purged from the server. This is necessary so that structures from the previous session are not re-used after the user has requested a change in session level, and so that "zombie" data structures and processes do not continue to utilize resources on the server after they have

out-lived their usefulness. Specifically, the following remote application support structures and processes should be purged at the end of a user session:

1. RAMSKT Map Database entry[5]
2. APSMSKT Map Database entry
3. User Database entry
4. MSKT Databases[5]
5. SSS Child processes
6. APS processes
7. TRSS Parent[5] and Child processes
8. RA processes

Furthermore, a TRSS Child process should not live longer than the remote application that it was invoked to serve. The same is true of SSS Child processes, which should not live longer than the APS processes that they were invoked to serve. This issue was discussed in Section B of this chapter.

Cleanup mechanisms are currently in the design phase, and are a major area for future work.

4. Unauthorized Channels

The RAMSKT Map and APSMSKT Map Databases are system-low databases, accessible to remote applications and application protocol servers running at all security levels. This is necessary because all RAs and APSs must be able to look up the handle of the MSKT Database to be used for their socket communications. However, because applications running at all security levels use the same database, there exists the possibility of a covert channel. This issue was identified in [3] and remains unresolved.

[5] The RAMSKT Map Database entry, MSKT Database used for RA communications, and TRSS Parent process should only be purged at the end of the user session if the user has no other open sessions at that session level.

5. Interactive Remote Application Support

The remote applications developed for this project were very simple programs meant to demonstrate proof-of-concept functionality of the remote application support mechanism. In a realistic environment, more sophisticated remote applications will be desired. This may include interactive and even graphical remote applications, both of which are unsupported by the current CGI remote application invocation mechanism. To enable the execution of interactive text-based remote applications, a telnet-like APS could take the place of the CGI web shell. Its role would be similar to that of the web shell: to relay input received from the user to a remote application on the server, and output from the remote application back to the user, but with the difference that its connection would be persistent. Thus, unlike the CGI web shell, it could relay communications back and forth between the application and user multiple times over the same socket connection. A special-purpose application would need to be developed on the MYSEA client side in addition to the server side in order to support this type of communication.

To enable graphical remote application support, a network-based windowing system such as X-Windows could be ported to the MYSEA environment. This is one of the more exciting prospects for future remote application support.

D. CONCLUSION

The need for high-assurance architectures that implement multi-domain information protection mechanisms is widespread and growing. However, such architectures will not be adopted unless they provide users with currently required functionality, the ability to easily incorporate new applications and software updates, and a familiar interface. The implementation of remote application support on the MYSEA server has contributed to the overall usability of the MYSEA architecture by allowing users logged in from diskless clients to execute server-resident applications using only a web browser. The use of remote applications increases the ease of application configuration and maintenance, and relieves constraints caused by the potentially limited amounts of RAM and lack of non-volatile storage available on the diskless clients. It is hoped that the development of high-assurance, highly usable MLS architectures such as

MYSEA will encourage the adoption of MLS computing systems by government, military, and business organizations that stand to benefit from their use of such systems.

APPENDIX A: SOURCE CODE LISTING

This appendix provides a listing of the internal MYSEA source code files that were modified or created for this project. Of these, the files implementing the Trusted Remote Session Server (TRSS), the swget client, and the CGI remote application invocation mechanisms were created from scratch. The files implementing the TFTP client were adapted from publicly available source code [14], but were new to the MYSEA distribution; these files are indicated with an asterisk. Files implementing various other MYSEA components, including the Trusted Path Server (TPS), Secure Session Server (SSS), RAMSKT Map Database, Source Address Binding Database, and Socket Handler Module, existed in previous MYSEA distributions but were modified during this project; these files are indicated with a double asterisk. Remote application support was integrated into the February 3, 2006 (1.1x) version of the MYSEA server software.

- TRSS files:
  `/usr/local/mysea/trss/`
  `/usr/local/mysea/trss/Makefile`
  `/usr/local/mysea/trss/trss.c`
  `/usr/local/mysea/trss/trss.h`

- CGI files:
  `/usr/local/mysea/cgi-bin/`
  `/usr/local/mysea/cgi-bin/ra_demo.cgi`
  `/usr/local/mysea/cgi-bin/simple_cmd_win.cgi`
  `/usr/local/mysea/cgi-bin/webshell.cgi`

- swget files:
  `/usr/local/mysea/swget/`
  `/usr/local/mysea/swget/Makefile`
  `/usr/local/mysea/swget/swget.c`

- TFTP client files:

  ```
  /usr/local/mysea/tftp/*
  /usr/local/mysea/tftp/aconfig.h*
  /usr/local/mysea/tftp/config.h*
  /usr/local/mysea/tftp/extern.h*
  /usr/local/mysea/tftp/main.c*
  /usr/local/mysea/tftp/Makefile*
  /usr/local/mysea/tftp/tftp.c*
  /usr/local/mysea/tftp/tftpsubs.c*
  /usr/local/mysea/tftp/tftpsubs.h*
  /usr/local/mysea/tftp/version.h*
  ```

- Other MYSEA files:

  ```
  /usr/local/mysea/include/ramskt_map.h**
  /usr/local/mysea/make_all.sh**
  /usr/local/mysea/make_tar**
  /usr/local/mysea/Makefile**
  /usr/local/mysea/sss/sss.c**
  /usr/local/mysea/test/test_socket_server.c**
  /usr/local/mysea/tps/tps_util.c**
  /usr/local/mysea/util/ramskt_map.c**
  /usr/local/mysea/util/sa_bind.c**
  /usr/local/mysea/util/skt_hndlr.c**
  ```

APPENDIX B: INSTALLATION PROCEDURES

The purpose of this appendix is to describe the installation procedures for the remote application support components developed during this project. These include the Trusted Remote Session Server (TRSS), the TFTP and swget clients, and the CGI remote application invocation mechanisms.

The following instructions include references to the Secure Attention Key (SAK). On the XTS-400 console, these are the "Alt" and "Print Screen" keys pressed simultaneously. When instructed to set the security and integrity levels, the user should issue a SAK followed by the *sl* command, then enter the desired security and integrity levels at the prompts.

The user should be logged in to the MSYEA server as admin for all of the following steps.

A. CONFIGURE MYSEA DAEMONS

For the purposes of remote application support, all active network interfaces must be configured at the level of the MLS LAN and serviced by a single daemon. This will affect the following steps within the NIC TCP/IP parameter configuration section of the normal MYSEA installation procedures:

1. In the *tcpip_edit* step, only the *tcpip_mls* daemon must be created. When asked whether to add another network interface entry for the daemon, type *y* and supply the parameters for each active network interface. If the *tcpip_mls* daemon has been previously configured to service only the MLS interface, it should be re-configured as follows:

   (Set security and integrity levels – min:max)
   SAK
   Enter command? `tcpip_edit`
   `change`
   `tcpip_mls` for Enter daemon name
   `<CR>` for Enter TCP/IP daemon description

<CR> for Modify the TCP/IP parameters
y for Modify network interface configuration
add
/dev/ether1 for Enter TCP/IP device name?
192.168.10.1 for Enter interface address?
0.0.0.0 for Enter destination address?
192.168.10.255 for Enter broadcast address?
255.255.255.0 for Enter network mask?
exit to return to the previous menu
<CR> for Modify the route configuration?
<CR> for Modify the resolver configuration?
exit – to leave tcpip_edit

(If configuring remote application support for a network architecture other than the test architecture described in Chapter V, repeat this process for each active network interface besides /dev/ether0 and /dev/ether1.)

2. In the *daemon_edit* step, only the *tcpip_mls* daemon must be created. If daemons such as *tcpip_nipr*, *tcpip_sipr*, or *tcpip_coin* have been previously configured to service single-level networks, they should be disabled as follows:

(Set security and integrity levels – min:max)
SAK
Enter command? daemon_edit
change
<daemon_name> for Enter daemon name
<CR> for all prompts but the following:
n for Start daemon at startup
exit – to leave daemon_edit

Repeat for each single-level daemon. A list of configured daemons may be retrieved by issuing the *list* command within *daemon_edit*. All but the *tcpip_mls*, *tps_udpd*, and *ssdd* daemons should be disabled.

3. In the *sda* step, each interface should be configured at the level of the MLS LAN (max:il3):

   (Set security and integrity levels – max:max)
   SAK
   Enter command? `sda`

Enter device?	`/dev/ether1`
Enter new device security level and categories?	`max`
Enter new device integrity level and categories?	`il3`
Modify discretionary access?	`n`
Is access correct?	`y`

   (If configuring remote application support for a network architecture other than the test architecture described in Chapter V, repeat this process for each active network interface besides /dev/ether0 and /dev/ether1.)

B. BUILD MYSEA BINARIES

The mysea.tar file contains already built object code and executables. Issuing the `make_all.sh` command during the normal MYSEA installation process will remove and regenerate all of the MYSEA executables, including the newly implemented remote application support components. The following instructions describe how to remove and regenerate these components individually.

(Set security and integrity levels – sl0:il3)
`run`

To rebuild the Trusted Remote Session Server (TRSS) executable:

```
cd /usr/local/mysea/trss
make clean
make
```

To rebuild the TFTP client executable:

```
cd /usr/local/mysea/tftp
make clean
make
```

To rebuild the swget client executable:

```
cd /usr/local/mysea/swget
make clean
make
```

C. CONFIGURE TRSS AS A TRUSTED PROGRAM

The TRSS executable must be designated a trusted program and granted specific privileges in order to function properly. This may be accomplished using the *tp_edit* program provided by the STOP 6 operating system. The following steps are required:

(Set security and integrity levels – min:max)
SAK
Enter command? `tp_edit`

`cd`
`cd`
`cd` – make sure in /trusted directory
`add`

`trss`	for name
`/usr/local/mysea/bin/trss`	for path
`<CR>`	for max integrity
`<CR>`	for min integrity
`y`	for assign privileges

`<CR>`	for all privileges but the following:
`y`	for Set owner/group
`y`	for Simple security exempt
`y`	for Security star property exempt
`y`	for Simple integrity exempt
`y`	for Integrity star property exempt
`y`	for Discretionary access exempt
`y`	for Trusted parent exempt

`exit` – to leave tp_edit

Now use *fsm* to set the permissions on the trusted executable:

(Set security and integrity levels – min:max)
SAK
Enter command? `fsm`

`cd`	
`/trusted`	for path
`change`	
`trss`	for name
`N`	for Modify access level
`<CR>`	for new owner & group
`y`	for discretionary access
`rwx`	for owner
`<CR>`	for username
`none`	for group
`<CR>`	for group name
`none`	for other
`N`	for display object
`Y`	for OK to change?

`exit` – to leave tp_edit

D. SET UP CGI SCRIPTS

The CGI remote application invocation scripts must be placed within the web server's cgi-bin directory. The following steps should be taken only after the /home/http directory has been created and populated during the course of the normal MYSEA installation:

(Set security and integrity levels – sl0:il3)

SAK

Enter command? `run`

```
cp /usr/local/mysea/cgi-bin/* /home/http/cgi-bin/
```

E. ENABLE DEBUGGING

Optionally, the TRSS Parent and Child processes may be configured to print debug statements to a log file by uncommenting the "DEBUG_OSS = -DDEMO" line within the TRSS Makefile:

(Set security and integrity levels – sl0:il3)

SAK

Enter command? `run`

```
cd /usr/local/mysea/trss
edit Makefile
```

Uncomment the line `"DEBUG_OSS = -DDEMO"`

The TRSS binary must now be re-compiled:

```
make clean
make
```

Additionally, the trusted TRSS binary must be replaced using *tp_edit*:

(Set security and integrity levels – min:max)

SAK

Enter command? `tp_edit`

`cd`

`cd`

`cd` – make sure in /trusted directory

`change`

Enter program name	`trss`
Replace program file [N]	`y`
Enter pathname:	`/usr/local/mysea/bin/trss`
Enter maximum integrity:	`<CR>`
Enter program name	`<CR>`
Enter minimum integrity	`<CR>`
Change privileges [N]	`<CR>`

`exit` – to leave tp_edit

Debugging logs will be created in the /tmp directory at the level of the TRSS processes (max:il3). Logs created by the TRSS Parent will have names of the form trsspar_X.tmp, where X is the process ID of the parent; logs created by the TRSS Child will have names of the form trsschild_Y.tmp, where Y is the process ID of the child.

Debugging may be disabled by re-commenting the appropriate line in the Makefile, re-compiling the TRSS executable, and re-running *tp_edit* to replace the trusted TRSS binary.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: TEST PROCEDURES

The purpose of this appendix is to document the test procedures used in the Test Plan presented in Chapter V. Steps should be taken to ensure that the following preconditions are met before testing:

- The test network connecting the MYSEA server, MYSEA client, and external single-level server is set up as illustrated in Chapter V, Figure 8 ("Test Network Topology").
- The mdemo1 account exists on the MYSEA server with default session level sl1:il3, maximum session level at least sl3:il3, and home directory /home/mdemo1/. (Following the standard MYSEA installation instructions will ensure this.)
- /home/mdemo1/ exists as a deflection directory on the MYSEA server. To create the directory, take the following steps:

  (As admin, set the security level to min:max)

  SAK

  `fsm`

`mkdir`	for Enter request
`/home/mdemo1`	for Enter the directory to create
`y`	for Should this be a deflection directory
`rwx`	for Enter directory modes for owner
`<CR>`	for Enter user name for specific permission
`rx`	for Enter directory modes for group
`<CR>`	for Enter group name for specific permission
`rx`	for Enter directory modes for others
`deflect`	for Enter request
`y`	for Disable deflection
`change`	for Enter request
`/home/mdemo1`	for Enter pathname
`n`	for Modify access level
`mdemo1`	for Enter new owner name
`other`	for Enter new group name
`n`	for Modify discretionary access

`n` for Display the object

`y` for Okay to change

`exit` – to leave fsm

- /home/mdemo1/ contains a text file entitled test.txt at security level sl1:il3. The file is readable by user mdemo1, and contains the single line:

  `Test file at sl1 il3 (SIM_UNCLASSIFIED).`

  Create this file as follows:

  (As mdemo1, set security and integrity levels – sl1:il3)

  SAK

  Enter command? `run`

  edit `/home/mdemo1/test.txt`

  Add the line specified above.

- The Allowed TPE, Source Address Binding, and Peer Level Databases on the MYSEA server are configured as described below. The user should edit the specified configuration files as admin while running a session at level sl0:il3.
  - The Allowed TPE Database configuration file (/usr/local/mysea/tpe_list) contains the following single entry:

# TCBE ID	Comment
`192.168.0.31`	`Java TPE`

  - The Source Address Binding Database configuration file (/usr/local/mysea/sa_bind) contains the following entries and no others:

`# DestinationIP`	`SourceIP`	`Netmask`
`192.168.0.0`	`192.168.0.38`	`255.255.255.0`
`192.168.10.0`	`192.168.10.1`	`255.255.255.0`

  - The Peer Level Database configuration file (/usr/local/mysea/peer_lvl) contains the following entries and no others, unless indicated otherwise by specific test procedures:

`# PeerIP`	`MLS_flag`	`PeerLevel`
`192.168.0.38`	`0`	
`192.168.10.1`	`1`	`sl1 il3`
`192.168.10.2`	`1`	`sl1 il3`

- The single-level server is running an externally accessible TFTP server, which contains in its /tftpboot directory the file tftp_test.txt. This file is world-readable and contains the following content:

```
Test file from 192.168.10.2.
```

  - To set up the TFTP server on the Red Hat Linux system plutodemo, take the following steps:
    1. Open /etc/xinetd.d/tftp
    2. Set “disable” to “no” (if it is not already)
    3. Click the Red Hat menu icon
    4. Select “System Settings”
    5. Select “Server Settings”
    6. Select “Services”
    7. Check “xinetd”
    8. Click “Start” (or “Restart” if the process is already running)
    9. Create tftp_test.txt in /tftpboot (if it does not exist already)

- The single-level server is running an externally accessible web server, which contains in its /var/www/cgi-bin directory a world-readable and world-executable CGI script named ra_demo.cgi. This script may be found in the cgi-bin directory contained within mysea.tar if it is not already installed on the single-level server. The server’s /var/www/html directory should *not* contain a file called nosuchfile.html.
  - To set up the web server on the Red Hat Linux system plutodemo, take the following steps:
    1. Click the Red Hat menu icon.
    2. Select “System Settings”
    3. Select “Server Settings”
    4. Select “Services”
    5. Check “httpd”
    6. Click “Start” (if the process is not already running)
    7. Place ra_demo.cgi in /var/www/cgi-bin (if it is not there already)
    8. Ensure that the /var/www/html directory does not contain a file called nosuchfile.html

Once these preconditions are met, testing may begin. The following test suites may be performed individually or in sequence. Unless otherwise indicated, the MYSEA

daemons should be running on the server prior to the start of each test. To start the daemons on the server:

1. As admin, set the security level to max:max.
2. SAK
3. Start the daemons:
   `startup`

The operating system should report that the *tcipip_mls*, *tps_udpd*, and *ssdd* daemons were started successfully.

For each of the following tests, unless otherwise indicated, the user should be logged in as mdemo1 via the Java TPE on the MYSEA client and have an active session running. The session level should remain at the default (SIM_UNCLASSIFIED, i.e., sl1:il3) unless a specific test step describes otherwise. To run a session from the client:

1. Launch the Java TPE application by double-clicking the “tcbe” icon on the desktop.
2. Set the remote IP address to that of the MYSEA server (192.168.0.38) and press Enter.
3. Click the “SAR” button.
4. At the login prompt, type “mdemo1” and press Enter.
5. At the password prompt, type the password for mdemo1 and press Enter.
6. Click the “SAR” button.
7. Type “run” and press Enter.
8. To access the web interfaces specified in the following test suites, launch the web browser on the client by double-clicking the “Netscape” icon on the desktop.

A. TRSS TESTING

Compile the test client and server on the MYSEA server, if necessary:

1. As admin, set the security level to sl0:il3 and issue the *run* command.

2. Navigate to the test directory:
```
cd /usr/local/mysea/test
```

3. Compile the test client:
```
make test_socket_client
```

4. Compile the test server:
```
make test_socket_ra
```

Initialize the connection between the test client and server:

From the MYSEA client:

5. Navigate to the MYSEA server web shell by entering `http://192.168.0.38/cgi-bin/webshell.cgi` in the "Address" field of the web browser.
6. Invoke the test server as a remote application by entering `/usr/local/mysea/test/test_socket_ra` in the "Command" field of the web page and clicking the "Enter" button.
   The web page should become blank while waiting for output from the remote application.

From the MYSEA server:

7. As admin, set the security level to max:il3 and issue the *run* command.
8. Issue the *startx* command. (This will ensure that results printed to the screen by the client are readable in full.)
9. Invoke the test client:
   `/usr/local/mysea/test/test_socket_client`
   A menu of commands should be displayed.
10. Select option Z and set both default IP addresses to 192.168.0.38.

11. Select option 1 to connect to the test server. Accept the default IP address to connect to (192.168.0.38). The client should report that a connection was established.
12. To perform test steps a1-a13, select the menu option corresponding to the type of testing specified for each step in the table below. Whenever prompted for an IP address, accept the default (192.168.0.38). Whenever prompted for a number of bytes, enter 32. When prompted for a port number, start with 2000 and increment by one for each port request.

The results for each test step should include the output provided in the "Expected Results Summary" column of the table below. For a complete record of the observed results for this test suite, see Appendix D.

Test ID	Menu selection	Port	Expected Results Summary
a1	2 - Do read testing	N/A	`error 0`
a2	3 - Do write testing	N/A	`error 0`
a3	4 - Do select testing - NON_BLOCKING	N/A	`error 0`
a4	5 - Do listen testing	2000	`error 0`
a5	6 - Do accept testing	2001	`error 0`
a6	7 - Do shutdown testing	2002	`error 0`
a7	8 - Do send testing	N/A	`error 0`
a8	9 - Do sendto testing	2003	`error 0`
a9	A - Do recv testing	N/A	`error 0`
a10	B - Do recvfrom testing	2004	`error 0`
a11	C - Do fork testing	N/A	`err 0`

Test ID	Menu selection	Port	Expected Results Summary
a12	D - Do Blocked I/O testing	2005	error 0
a13	E - Do Misc testing	2006	---fcntl(F_GETFD) - 0--- error 0 ---fcntl(F_GETFL) - 2--- error 0 ---fcntl(F_SETFD) - 0--- error 0 ---fcntl(F_SETFL) - 0--- error 0 ---getpeername - 192.168.0.38--- error 0 ---getsockname - 192.168.0.38--- error 0 ---getsockopt(KEEPALIVE) - 0--- error 0 ---setsockopt(KEEPALIVE) - 1--- error 0 ---ioctl – FIONREAD - 0--- error 0

From the MYSEA client:

13. After the tests have been completed, verify that the web page is still blank, indicating that the remote application server is still running. Had the TRSS Child been unable to handle any of the server's MSKT socket calls, both processes would have terminated and a new web shell would have been displayed.

From the MYSEA server:

14. Select menu option 0 to exit. The client should report, "Test is complete."

From the MYSEA client:

15. Verify that the browser window once again displays the web shell, indicating that the remote application server has terminated.

B. WEB SHELL FUNCTIONAL TESTING

From the client TPE, verify that the current session level is "SIM_UNCLASSIFIED" by pressing the "SAR" button and issuing the "session" command.

Navigate to http://192.168.0.38/cgi-bin/webshell.cgi. Enter each of the following commands in the "Command" field, clicking "Enter" after each one. The expected results listed in the table below will appear embedded within an emulated terminal window.

Test ID	Command	Expected Result
b1	ls	test.txt
b2	cd /var	(New command prompt: [mdemo1 /var]#)
	ls	cache lib log run spool tmp
b3	whoami	mdemo1
b4	/usr/local/mysea/tools/mysea_level	SIM_UNCLASSIFIED

Close the browser window before proceeding to the next test suite.

C. WEB SHELL EXCEPTION TESTING

Open a new browser window and navigate to http://192.168.0.38/cgi-bin/webshell.cgi. Issue the following commands, clicking the "Enter" button after each one.

Test ID	Command	Expected Result
c1	fakecmd	No such file or directory
	ls	test.txt
c2	(None – click "Enter")	(New shell; no new output)
	ls	test.txt
c3	(Follow instructions below.)	(New shell; no new output)
	ls	test.txt

For test c3, part 1:

1. Save the web shell as an .html file. Open the file in a text editor and make the following changes:
    a. Within the `<form>` element, add:
    `action=http://192.168.0.38/cgi-bin/webshell.cgi`
    b. Within the `<input>` element, change `cmd` to `fakeinputname`.
2. Open the modified file in the web browser.
3. Type `ls` in the "Command" field, and click "Enter".

D. COMMAND MENU FUNCTIONAL TESTING

Navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Select each of the following menu options, clicking the "Execute" button after each one. An empty "Arg" entry in the table below indicates that the default input value should be deleted from the text field and left empty. Otherwise, the default value should be replaced with the value indicated in the table. A grayed-out "Arg" entry signifies that the command menu does not have a corresponding text field.

The expected results listed below will be displayed below the "Current directory" indicator, except in the case of the web page request in test d6. In this case, the web page

will replace to command menu. To return to the command menu for the test d7, click the web browser's "Back" button.

Test ID	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
d1	"List contents of directory"				Contents of directory: test.txt
d2	"Display contents of file"	test.txt			Contents of file test.txt: Test file at sl1 il3 (SIM_UNCLASSIFIED).
d3	"Change to directory"	/var			("Current directory" indicator lists "/var" as the current directory.)
	"List contents of directory"				Contents of directory: cache lib log run spool tmp
d4	"Change to directory"				("Current directory" indicator lists "/home/mdemo1" as the current directory.)
	"Move file from"	test.txt	test1.txt		Moving file from test.txt to test1.txt
	"List contents of directory"				Contents of directory: test1.txt
d5	"Copy file from"	test1.txt	test.txt		Copying file from test1.txt to test.txt
	"List contents of directory"				Contents of directory: test.txt test1.txt
d6	"Retrieve web page from"	http://192.168.10.2/cgi-bin/ra_demo.cgi			(See below.)

Test ID	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
d7	"TFTP GET"	tftp_test.txt	192.168.10.2	/tmp/test_d7.txt	tftp> Received 30 bytes in 0.1 seconds [1665 bit/s] (bit rate may differ)
	"Display contents of file"	/tmp/test_d7.txt			Contents of file /tmp/test_d7.txt: Test file from 192.168.10.2.

The output from test d6 should be a web page containing the following text. (The "Current time" value will differ.)

Demonstration of MYSEA Remote Application Support

This is a sample web page.

Web page request received from 192.168.10.1.
Web page served by 192.168.10.2.
Current time is Tue Mar 7 09:44:01 PST 2006.

After completing this test suite, delete the file test1.txt from /home/mdemo1, either via the web shell or directly on the MYSEA server.

E. COMMAND MENU EXCEPTION TESTING

Navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Select the menu option listed for each test in the table below, clicking the "Execute" button after each one. For an entry of "(None)", click the "Execute" button without selecting any radio button.

Test ID	Menu Selection	Arg 1	Arg 2	Arg 3	Expected Result
e1	(Follow instructions below.)				Illegal input: fakecmd
	"List contents of directory"				Contents of directory: test.txt
e2	(None)				(New command menu.)
	"List contents of directory"				Contents of directory: test.txt
e3	(Follow instructions below.)				(New command menu.)
	"List contents of directory"				Contents of directory: test.txt
e4	"List contents of directory"	a;b			Illegal input: ls a;b
	"List contents of directory"				Contents of directory: test.txt
e5	"Move file from"				Moving file from to mv: missing file argument Try `mv --help' for more information.
	"List contents of directory"				Contents of directory: test.txt

For test e1, part 1:

1. Save the command menu as an .html file. Open the file in a text editor and make the following changes:

a. Within the `<form>` element, add: `action=http://192.168.0.38/cgi-bin/simple_cmd_win.cgi`

b. Within the `<input type=radio name=cmd value='ls' …>` element, change `ls` to `fakecmd`.

2. Open the modified file in the web browser.

3. Select the "List contents of directory" option with no argument, and click “Execute.”

For test e3, part 1:

1. Save the command menu as an .html file. Open the file in a text editor and make the following changes:

a. Within the `<form>` element, add: `action=http://192.168.0.38/cgi-bin/simple_cmd_win.cgi`

b. Within the `<input type=radio name=cmd value='ls' …>` element, change `cmd` to `fakeinputname`.

2. Open the modified file in the web browser.

3. Select the "List contents of directory" option with no argument, and click “Execute.”

F. TFTP CLIENT FUNCTIONAL TESTING

Navigate to http://192.168.0.38/cgi-bin/webshell.cgi. Issue the following commands, clicking the “Enter” button after each one.

Test ID	Command	Expected Result
f1	echo getq tftp_test.txt /tmp/test_f1.txt | /usr/local/mysea/tftp/tftp 192.168.10.2	tftp> Received 30 bytes in 0.0 seconds [5336 bit/s] (bit rate may differ)
	cat /tmp/test_f1.txt	Test file from 192.168.10.2.

G. WEB CLIENT FUNCTIONAL TESTING

Navigate to http://192.168.0.38/cgi-bin/webshell.cgi. Issue the following commands, clicking the "Enter" button after each one.

Test ID	Command	Expected Result
g1	/usr/local/mysea/swget/swget http://192.168.10.2/cgi-bin/ra_demo.cgi	(See below.)
g2	/usr/local/mysea/swget/swget 192.168.10.2/cgi-bin/ra_demo.cgi	(See below.)

For each of these tests, the following HTML markup should be displayed. (The "Current time" value will differ.)

```
<html>
<head><title>Remote Application Support Demo</title></head>
<body>
<table border=0 cellpadding=2 cellspacing=0 width=800>
<tr><td><table border=1 cellpadding=4 cellspacing=0 width=100%>
<tr><td>
<table border=0 cellpadding=2 cellspacing=0 width=100%>
<tr bgcolor=#88bbee align=center>
<td><font size=+3><b>
Demonstration of MYSEA Remote Application Support
</b></font></td></tr></table>
</td></tr></table>
```

```
</td></tr>
<tr><td> <td><tr>
<tr align=center><td>
<font size=+1>This is a sample web page.</font>
</td></tr>
<tr><td> <td><tr>
<tr align=center><td>
<font size=+1>Web page request received from 192.168.10.1.</font>
</td></tr>
<tr align=center><td>
<font size=+1>Web page served by 192.168.10.2.</font>
</td></tr><tr align=center><td>
<font size=+1>Current time is Tue Mar 7 10:20:21 PST 2006
.</font>
</td></tr></table>
</body>
</html>
```

H. WEB CLIENT EXCEPTION TESTING

Navigate to http://192.168.0.38/cgi-bin/webshell.cgi. Issue the following commands, clicking the "Enter" button after each one.

Test ID	Command	Expected Result
h1	/usr/local/mysea/swget/swget	Usage: swget [URL]
h2	/usr/local/mysea/swget/swget arg1 arg2	Usage: swget [URL]
h3	/usr/local/mysea/swget/swget 1.1.1.1	Could not connect. swget: connect: Permission denied
h4	/usr/local/mysea/swget/swget http://192.168.10.2/nosuchfile.html	(See below.)

For test h4, the following HTML markup should be displayed:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /nosuchfile.html was not found on this server.</p>
<hr />
<address>Apache/2.0.40 Server at <a
href="mailto:root@localhost">192.168.10.2</a> Port 80</address>
</body></html>
```

I. ACCEPTANCE TEST CASE 1 FUNCTIONAL TESTING

Navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Execute the following commands, clicking the “Execute” button after each one.

Test ID	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
i1	"TFTP GET"	tftp_test.txt	192.168.10.2	/tmp/test_i1.txt	tftp> Received 30 bytes in 0.0 seconds [8677 bit/s] (bit rate may differ)
	"Display contents of file"	/tmp/test_i1.txt			Contents of file /tmp/test_i1.txt: Test file from 192.168.10.2.

J. ACCEPTANCE TEST CASE 1 EXCEPTION TESTING

The following tests involve the re-configuration of the Peer Level Database on the MYSEA server. For each test, take the following steps:

Close all open browser windows on the MYSEA client.

On the MYSEA server:

(As admin, set security and integrity levels – sl0:il3)

SAK

Enter command? `run`

edit `/usr/local/mysea/peer_lvl`

Configure the peer level of the host 192.168.10.2 to the value specified in the "Peer Level" table below. If the value is "Undefined," comment the entry out of the database file by adding a # to the beginning of the line.

(Set security and integrity levels – max:max)

SAK

`stop_daemon`

`ssdd` for Enter daemon name

`tps_udpd` for Enter daemon name

`<CR>` – to leave stop_daemon

SAK

`start_daemon`

`tps_udpd` for Enter daemon name

`ssdd` for Enter daemon name

`<CR>` – to leave start_daemon

Now log in as mdemo1 from the MYSEA client, run a session at the default session level (sl1:il3), open a browser window, and navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Make the menu selection indicated in the table and click the "Execute" button.

Test ID	Peer Level	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
j1	sl2:il3	"TFTP GET"	tftp_test.txt	192.168.10.2	/tmp/test_j1.txt	tftp: sendto: Permission denied tftp>
		"Display contents of file"	/tmp/test_j1.txt			Contents of file /tmp/test_j1.txt:

Test ID	Peer Level	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
j2	sl0:il3	"TFTP GET"	tftp_test.txt	192.168.10.2	/tmp/test_j2.txt	tftp: sendto: Permission denied tftp>
		"Display contents of file"	/tmp/test_j2.txt			Contents of file /tmp/test_j2.txt:
j3	Undefined	"TFTP GET"	tftp_test.txt	192.168.10.2	/tmp/test_j3.txt	tftp: sendto: Permission denied tftp>
		"Display contents of file"	/tmp/test_j3.txt			Contents of file /tmp/test_j3.txt:

Close all browser windows, set the peer level for 192.168.10.2 back to sl1:il3 in the Peer Level Database, and restart the MYSEA daemons before proceeding to the next test suite.

K. ACCEPTANCE TEST CASE 2 FUNCTIONAL TESTING

For test k1, log in from the TPE as mdemo1 and run a session at the default session level (sl1:il3). Then, navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi, make the menu selection indicated in the table entry below, and click “Execute.”

To avoid cleanup issues, reboot the MYSEA server and restart the daemons between tests k1 and k2.

For test k2, log in from the TPE as mdemo1, then click the “SAR” button and issue the “sl” command to change the session level to SIM_CONFIDENTIAL. Run a session, then open a new browser window and navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Make the menu selection indicated in the table entry below and click “Execute.”

Test ID	Session Level	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
k1	sl1:il3	"Retrieve web page from"	http://192.168.0.38/ cgi-bin/ra_demo.cgi			(See below.)
k2	sl3:il3	"Retrieve web page from"	http://192.168.0.38/ cgi-bin/ra_demo.cgi			(See below.)

The output from test k1 should be a web page containing the following text. (The "Current time" value will differ.)

Demonstration of MYSEA Remote Application Support

This is a sample web page.

Web page request received from 192.168.0.38.
Web page served by 192.168.0.38.
Current time is Tue Mar 7 07:35:36 PST 2006.

Process user name is mdemo1.
Process session level is sl1 il3 (SIM_UNCLASSIFIED).

The output from test k2 should be a web page containing the following text. (The "Current time" value will differ.)

Demonstration of MYSEA Remote Application Support

This is a sample web page.

Web page request received from 192.168.0.38.
Web page served by 192.168.0.38.
Current time is Tue Mar 7 07:41:36 PST 2006.

Process user name is mdemo1.
Process session level is sl3 il3 (SIM_CONFIDENTIAL).

L. ACCEPTANCE TEST CASE 2 EXCEPTION TESTING

This test involves the re-configuration of the Peer Level Database. Perform the steps listed in test j to undefine the peer level of host 192.168.0.38 in the Peer Level Database and restart the MYSEA daemons.

Log in as mdemo1 via the TPE and run a session at the default session level. Navigate to http://192.168.0.38/cgi-bin/simple_cmd_win.cgi. Select the menu item indicated in the table entry below and click the “Execute” button.

Test ID	Session Level	Menu selection	Arg 1	Arg 2	Arg 3	Expected Result
l1	Undefined	"Retrieve web page from"	http://192.168.0.38/ cgi-bin/ra_demo.cgi			Could not connect. swget: connect: Permission denied

For future remote application support, restore the Peer Level Database to its original configuration after the completion of this test.

APPENDIX D: TRSS DEVELOPMENTAL TESTING RESULTS

A complete record of the observed TRSS developmental testing results is provided in the table below. Observed results for all other test suites were identical to the expected results listed within the testing procedures of Appendix C.

Test results may differ slightly from the observed results listed below due to random process ID assignment and timing issues, but in each case the client should report that the test was completed without errors (i.e., with the reported error equal to 0).

Test ID	Menu selection	Port	Observed Result
a1	2 - Do read testing	N/A	READ, error 0, buffer length 13 ---read 32 bytes---
a2	3 - Do write testing	N/A	read 32 bytes ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF WRITE, error 0, buffer length 14 ---write 32 bytes---
a3	4 - Do select testing - NON_BLOCKING	N/A	SELECT NB, error 0, buffer length 13 ---read 30 bytes---
a4	5 - Do listen testing	2000	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 Connected to 192.168.0.38, port 2000 Selecting - 3, 4- select() returns 1- result is 4 LISTEN, error 0, buffer length 28 ---Connection from 192.168.0.38---
a5	6 - Do accept testing	2001	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 Waiting for a connection Selecting - 3, 4- select() returns 1- result is 4 Connection from 192.168.0.38 ACCEPT, error 0, buffer length 36 ---Connected to 192.168.0.38, port 2001---
a6	7 - Do shutdown testing	2002	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 Connected to 192.168.0.38, port 2002 Selecting - 3, 4- select() returns 1- result is 4 SHUTDOWN, error 0, buffer length 41 ---Shutdown(RD) connection from 192.168.0.38---

Test ID	Menu selection	Port	Observed Result
a7	8 - Do send testing	N/A	read 32 bytes ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF SEND, error 0, buffer length 14 ---write 32 bytes---
a8	9 - Do sendto testing	2003	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 calling recvfrom Selecting - 3, 4- select() returns 1- result is 4 recvfrom 32 bytes Connection from 192.168.0.38 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEF SENDTO, error 0, buffer length 15 ---sendto 32 bytes---
a9	A - Do recv testing	N/A	RECV, error 0, buffer length 13 ---read 32 bytes---
a10	B - Do recvfrom testing	2004	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 calling sendto sendto 32 bytes RECVFROM, error 0, buffer length 36 ---recvfrom 32 bytes, from 192.168.0.38---
a11	C - Do fork testing	N/A	do_fork - read 41 bytes --in PARENT - PID 27421, child PID is 1911 -- do_fork - read 127 bytes -- -- get_result - read() only got 141 of 268 cmd 0, err 0, buffer EMPTY

Test ID	Menu selection	Port	Observed Result
a12	D - Do Blocked I/O testing	2005	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 Connected to 192.168.0.38, port 2005 Blocked I/O, error 0, buffer length 17 ---fcntl() succeeded--- Blocked I/O, error 0, buffer length 15 ---read() 30 bytes--- Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 1024 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 2048 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 3072 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 4096 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 5120 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read 6144 so far Selecting - 3, 4- select() returns 1- result is 4 do_blockedio - read() only got 0 of 1024 do_blockedio - total read 6144 Blocked I/O, error 0, buffer length 18 ---write() 6144 bytes---

Test ID	Menu selection	Port	Observed Result
a13	E - Do Misc testing	2006	IP address is 192.168.0.38 sin.addr is 0x2600a8c0 Connected to 192.168.0.38, port 2006 Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 18 ---fcntl(F_GETFD) - 0--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 18 ---fcntl(F_GETFL) - 2--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 18 ---fcntl(F_SETFD) - 0--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 18 ---fcntl(F_SETFL) - 0--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 26 ---getpeername - 192.168.0.38--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 26 ---getsockname - 192.168.0.38--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 25 ---getsockopt(KEEPALIVE) - 0--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 25 ---setsockopt(KEEPALIVE) - 1--- Selecting - 3, 4- select() returns 1- result is 3 Miscellaneous, error 0, buffer length 17 ---ioctl – FIONREAD - 0--- Selecting - 3, 4- select() returns 1- result is 3 Current test all done, error 0, buffer EMPTY

It should be noted that the output of the test client was inconsistent for test a13 ("Miscellaneous Testing"). Specifically, the client would sometimes only print a partial result summary before concluding the test and printing a new command menu. It is believed that this behavior was the result of the *mskt_ioctl* call added for the purpose of TRSS testing. Because the test client was not adapted to take into account the consequences of this call (merely to check that it returned a result of 0), the addition of the call interfered with the normal operation of the client. It was determined through the examination of TRSS log files that the TRSS Child process was handling the *mskt_ioctl* call correctly; the abnormal behavior was on the part of the test client. Furthermore, whenever the client reported results at least through the "`---ioctl – FIONREAD`" line, the results reported for the *mskt_ioctl* call and all previous calls indicated a return value of 0, constituting a successful result for the purposes of TRSS testing.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. National Security Agency. "The GIG Vision Enabled by Information Assurance." Available: http://www.nsa.gov/ia/industry/gig.cfm?MenuID=10.3.2.2. Accessed: 3/19/2006.

2. Irvine, C., Levin, T., Nguyen, T., Shifflett, D., Khosalim, J., Clark, P., Wong, A., Afinidad, F., Bibighaus, D., and Sears, J. *Overview of a High Assurance Architecture for Distributed Multilevel Security,* Proceedings 5th IEEE Systems, Man and Cybernetics Information Assurance Workshop, United States Military Academy, West Point, NY, 10-11 June 2004, pgs. 38-45.

3. Cooper, R. *Remote Application Support in a Multilevel Environment,* Naval Postgraduate School, Master's Thesis, March 2005.

4. Bell, D. E. and La Padula, L. J. *Secure Computer System: Unified Exposition and Multics Interpretation*, ESD-TR-75-306, MITRE Corporation, Bedford, MA, 1976.

5. Biba, K .J. *Integrity Considerations for Secure Computer Systems*, ESD-TR-76-372, MITRE Corporation, Bedford, MA, 1977.

6. DigitalNet. *XTS-400 Trusted Computer System Technical Overview*, Herndon, VA, 2002-2003. Available: http://www.digitalnet.com/solutions/information_assurance/pdf/XTS400%20Technical%20Description%206-8-04.PDF. Accessed: 11/27/2005.

7. Common Criteria Portal. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model, Version 2.2, Revision 256 – CCIMB2004-01-001*, January 2004. Available: http://www.commoncriteriaportal.org/public/files/ccpart1v2.2.pdf. Accessed: 11/27/2005.

8. National Information Assurance Partnership (NIAP). "XTS-400™ / STOP™ 6.1.E," March 2005. Available: http://niap.nist.gov/cc-scheme/st/ST_VID3012a.html. Accessed: 1/22/2006.

9. Eads, B. *Developing a High Assurance Multilevel Mail Server,* Naval Postgraduate School, Master's Thesis, March 1999.

10. Everette, T. *Examination of the Internet Message Protocol (IMAP) to Facilitate User-Friendly Multilevel Email Management,* Naval Postgraduate School, Master's Thesis, September 2000.

11. Brown, E. *Facilitating Secure Mail in a High Assurance LAN*, Naval Postgraduate School, Master's Thesis, September 2000.

12. Bersack, E. L. *Implementation of a Hypertext Transfer Protocol Server on a High Assurance Multilevel Secure Platform*, Naval Postgraduate School, Master's Thesis, December 2000.

13. Clark, P. *Policy-Enhanced Linux*, Proceedings 23rd National Information Systems Security Conference, Volume I, pp. 418-432, Baltimore, MD, October 2000. Available: http://cisr.nps.navy.mil/downloads/00paper_linux.pdf. Accessed: 12/06/2005.

14. Softpedia. "tftp-hpa 0.41." Available: http://linux.softpedia.com/progDownload/tftp-hpa-Download-5440.html. Accessed: 1/1/2006.

15. BryerJoyner, S. and Heller, D. *Secure Local Area Network Services for a High Assurance Multilevel Network*, Naval Postgraduate School, Master's Thesis, March 1999.

16. Gamma Group. "Gamma Web Shell." Available: http://legacy.gammacenter.com/gamma.py/products/WebShell. Accessed: 2/21/2006.

17. Center for Information Systems Security Studies and Research, Naval Postgraduate School. "Module Test Plan for MYSEA Version 2.0 (DRAFT)," March 2006.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Hugo A. Badillo
   NSA
   Fort Meade, MD

4. George Bieber
   OSD
   Washington, DC

5. RADM Joseph Burns
   Fort George Meade, MD

6. John Campbell
   National Security Agency
   Fort Meade, MD

7. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

8. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

9. Louise Davidson
   National Geospatial Agency
   Bethesda, MD

10. Steve Davis
    NRO
    Chantilly, VA

11. Vincent J. DiMaria
    National Security Agency
    Fort Meade, MD

12. CDR James Downey
NAVSEA
Washington, DC

13. Dr. Diana Gant
National Science Foundation

14. Jennifer Guild
SPAWAR
Charleston, SC

15. Richard Hale
DISA
Falls Church, VA

16. CDR Scott D. Heller
SPAWAR
San Diego, CA

17. Wiley Jones
OSD
Washington, DC

18. Russell Jones
N641
Arlington, VA

19. David Ladd
Microsoft Corporation
Redmond, WA

20. Dr. Carl Landwehr
DTO
Fort George T. Meade, MD

21. Steve LaFountain
NSA
Fort Meade, MD

22. Dr. Greg Larson
IDA
Alexandria, VA

23. CAPT Deborah McGhee
Headquarters U.S. Navy
Arlington, VA

24. Dr. Vic Maconachy
NSA
Fort Meade, MD

25. Doug Maughan
Department of Homeland Security
Washington, DC

26. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

27. John Mildner
SPAWAR
Charleston, SC

28. Mark T. Powell
Federal Aviation Administration
Washington, DC

29. Jim Roberts
Central Intelligence Agency
Reston, VA

30. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

31. Charles Sherupski
Sherassoc
Round Hill, VA

32. Dr. Ralph Wachter
ONR
Arlington, VA

33. David Wirth
N641
Arlington, VA

34. CAPT Robert Zellmann
CNO Staff N614
Arlington, VA

35. CDR Wayne Slocum
SPAWAR
San Diego, CA

36. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

37. Thuy D. Nguyen
Naval Postgraduate School
Monterey, CA

38. Melissa K. Egan
Civilian, Naval Postgraduate School
Monterey, CA